Ricardo Alexandre do Rosário Ribeiro

Licenciado em Engenharia Informática

Protein docking GPU acceleration

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador:	Hervé Paulino, Professor Auxiliar, FCT-UNL
Co-orientador:	Ludwig Krippahl, Professor Auxiliar, FCT-UNL

Júri

Presidente:	Professor Doutor João Manuel dos Santos Lourenço
Arguentes:	Professor Doutor Vasco Fernando de Figueiredo Tavares Pedro
Vogais:	Professor Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2019

Protein docking GPU acceleration

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Pela minha família, pátria e amigos.

Agradecimentos

Em primeiro lugar gostaria de agradecer à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa por estes cinco anos e meio, em especial pelo desafio envolvente no decorrer do curso e pelo que aprendi.

De seguida gostaria de agradecer aos meus orientadores, Professor Doutor Hervé Paulino e Professor Doutor Ludwig Krippahl pela oportunidade que me deram para desenvolver esta dissertação. Ao professor Hervé Paulino agradeço em especial o esclarecimento de diversas questões relacionadas com a *framework Marrow* assim como de programação em GPUs. Agradeço ao professor Ludwig Krippahl pelo apoio prestado em esclarecimentos de questões sobre o algoritmo BiGGER e respetiva implementação assim como o esclarecimento de questões relacionadas com *docking* entre proteínas.

Gostaria também de agradecer aos meus colegas de curso pelo apoio prestado desde o primeiro ano até ao último.

Por fim, gostaria de agradecer em especial à minha família pelo apoio prestado ao longo da minha vida. A eles dedico o esforço realizado para a dissertação, esperando que corresponda com a luta constante que temos feito em prol de um futuro melhor para nós. Também dedico este esforço aos meus familiares que já partiram assim como aos que hão de vir e que um dia, se Deus quiser, hão de estar a trabalhar nas suas dissertações.

Resumo

Na área científica da Bioinformática, determinar com precisão o complexo formado pela interação entre um par de proteínas é um problema computacionalmente difícil. Existem métodos e algoritmos para simular a fusão de um par de proteínas que demoram horas para executar a simulação recorrendo apenas à CPU, sendo em termos de trabalho/tempo ineficiente. Um desses métodos é o BiGGER, desenvolvido pelo prof. Nuno Palma et al. Este algoritmo assume características que lhe dão uma complexidade temporal inferior aos demais, pelo que os tempos de execução do BiGGER são menores do que a maior parte dos algoritmos e respetivos programas de *docking*. O estudo das interações entre proteínas tem aplicações medicinais, onde são desenvolvidas formas de proteger o Homem de doenças neuronais assim como permite desenvolvimentos na área do desenho e conceção de drogas assistido por computador. Para resolver a ineficiência referida, as ferramentas para *docking* foram optimizadas para usar a GPU como auxiliar na execução das simulações, reduzindo o tempo de execução de um cenário que dure horas para minutos ou até mesmo segundos. O presente documento aborda uma proposição para a paralelização do algoritmo BiGGER.

A implementação será feita recorrendo a técnicas de computação acelerada i.e. utilizar a GPU da máquina em que corre o algoritmo para auxiliar a CPU na computação que é necessária. Tendo mais recursos à disposição, é esperado que o tempo de execução do BiGGER baixe drasticamente por consequência do aumento significativo de performance face à versão sequencial. Em caso de sucesso, a versão futura do BiGGER poderá vir a ter vantagens adicionais face aos seus concorrentes. Por consequência deste aumento de performance, a nova versão do BIGGER permite oferecer uma proposta de valor ao utilizador, podendo este último dispor de uma ferramenta de trabalho eficiente no estudo das interações entre as proteínas em qualquer máquina pessoal ou profissional.

Palavras-chave: proteínas, *docking*, computação acelerada, GPU, Bioinformática, BiGGER

Abstract

In Bioinformatics, finding the complex resulting from an interaction between a pair of proteins is a computationally demanding task. There are methods and algorithms that simulate the binding between two proteins. However, the computation related to docking simulation has many extensive and repeating steps. Thus, the execution of the simulation if the program is CPU-only can last for hours, making the option of using these programs a very inefficient one in terms of work/time.

One of the methods used is BiGGER, created by prof. Nuno Palma and others. This algorithm has features that give it a lower time complexity compared to others, therefore execution times in BiGGER can be lower than most of the algorithms fitted for execution of docking simulations. Studying protein interactions has medical aplications, contributing to the development of ways to protect, diagnose and heal humanity from neuronal diseases. It also contributes to computer assisted drug design and development. To improve the execution time of docking programs, these were optimized for GPU execution, reducing the execution time in docking scenarios that can last for hours to minutes or even seconds.

This document presents an aproach to the implementation of optimizations to BiGGER, running it with GPU assistance. This implementation is to be done via high performance computing techniques, so that the machine's GPU assists the CPU on parallelizing those required computations. By having more resources at disposal, it should be expected that the execution time of BiGGER is reduced due to the improvement of BiGGER performance in relation to the sequential version.

If the implementations succeed, there will be additional advantages for BiGGER in relation to other algorithms. Thus, a value proposition is given for those who intend to use BiGGER as a method for efficiently studying interactions between proteins in a personal ou professional computer.

Keywords: proteins, docking , high performance computing, GPU, Bioinformatics, BiGGER

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

Ångstrom	Unidade padrão para a distância em biologia molecular, sendo que 1Å é equivalente a 0.1 nanómetros.
assessment	Termo inglês para estudar ou analisar e tomar conclusões sobre uma dada matéria.
bottleneck	Zona de código de um dado programa onde é verificado que atrasa a execução deste.
complementaridade	Princípio para descrever como é que duas entidades conseguem se unir.
conformação	Sinónimo de ajustamento.
core	Palavra de origem inglesa para definir a região central de uma entidade.
correlação	Relação geométrica entre duas posições num plano.
cristalografia de raios X	Método cientifico usado para elucidação das estruturas das proteínas. A cristalização de proteínas foi descoberta acidentalmente no século XIX (1840), por Hunfeld, antes da descoberta histórica dos raios X por parte de Wilhelm Röntgen (1895).
deprecated	Da lingua inglesa para caracterizar algo que é usável mas foi considerado como obsoleto, não devendo ser utilizado.
digitalização	A digitalização é o ato de converter algo físico em digital. No contexto da tese a digitalização diz respeito à etapa de converter a superfície molecular de uma proteína para um conjunto de grelhas tridimensionais.
embaraçosamente paralelo(a)	Termo usado para problemas ou computações que podem ser separados em tarefas independentes.
hotspot	Zona de código de um dado programa onde a proporção de instruções executadas é maior ou onde a fração de tempo de execução é maior.
kernel	Função que é programada para ser executada por uma ou mais unidades de processamento de gráficos.

ligando	Papel desempenhado pela proteína ou droga que toma a iniciativa de se juntar a uma outra proteína.
paralelização	Tarefa de paralelizar algo, sendo que no contexto da tese este termo é aplicado à otimização do algoritmo BiGGER..
receptor	Papel desempenhado pela proteína que ao contrário do ligando não toma a iniciativa de se juntar a outra.
surface	Palavra de origem inglesa para definir a região de superfície de uma entidade.

Siglas

API	Application Programming Interface.
APOD	Assess, Parallelize, Optimize, Deploy.
BiGGER	Bimolecular complex Generation with Global Evaluation and Ranking.
BoGIE	Boolean Geometric Interaction Evaluation.
CPU	Central Processor Unit.
CUDA	Compute Unified Device Architecture.
FFT	Fast Fourier Transform.
GPC	GPU Processing Cluster.
GPGPU	General-purpose computation on graphics proces- sing units.
GPU	Graphics Processor Unit.
GSC	Grid-based Surface Complementarity.
IDE	Integrated Development Environment.
MPI	Message Passing Interface.
NVCC	NVIDIA CUDA Complier.
PDB	Protein Database.
PDI	Protein-Drug Interaction.
PPI	Protein-Protein Interaction.
PSC	Pair-based Surface Complementarity.
RSMD	Root-mean-square deviation.

SM Stream Multiprocessor.
SP Scalar Processor.

VdW Van der Waals.

XOR Exclusive Or.

Capítulo 1

Introdução

1.1 Enquadramento e motivação

As proteínas não desempenham as suas funções de forma isolada, de acordo com Gonzalez e Kahn (2012) [10], estas interagem não só com outras proteínas como também com outros tipos de moléculas, como por exemplo ADN ou moléculas constituintes das drogas. Desta forma os mecanismos que determinam o estado de saúde de um organismo são controlados pelas interações entre proteínas.

Por sua vez o estudo destas interações tem garantido avanços na elucidação das formas moleculares associadas às doenças, trazendo avanços na proteção, diagnóstico e tratamento de doenças consideradas incuráveis. Um exemplo a considerar foi em 2018, um investigador português ter descoberto que a interação entre as proteínas S100B e beta-amilóide provocam um atraso na formação dos agregados do beta-amilóide, trazendo como beneficio a proteção contra a doença de Alzheimer [31]. Para além de avanços no estudo das doenças, o estudo tem permitido avanços consideráveis no desenho de drogas assistido por computador [40], permitindo a conceção de novas variantes de produtos farmacêuticos.

No entanto este estudo é computacionalmente pesado, o procedimento envolve uma fase de pesquisa exaustiva sobre o conjunto total de estruturas possíveis para o complexo de proteínas final a partir de um número elevado de rotações e conformações. O número de possibilidades cresce exponencialmente com o tamanho dos elementos do par [15].

Apesar de ser um processo computacionalmente pesado, este encontra-se dividido em etapas que são boas candidatas para execução em paralelo. Neste contexto, as unidades de processamento gráfico (GPUs) apresentam-se como uma boa solução para aumentar o desempenho da computação associada ao *docking* entre proteínas, sendo uma solução com custos financeiramente viáveis.

O presente documento aborda a implementação de acelerações ao algoritmo BiGGER [41], em que a GPU será utilizada para a paralelização das zonas de código onde a execução do BiGGER passa mais tempo, melhorando os tempos de execução do algoritmo. Adicionalmente o documento aborda a adição de novas funcionalidades necessárias ao *Marrow* para implementar as acelerações.

1.2 Conceito de *docking*

De acordo com Halperin et al (2002) [15], *docking* pode ser visto como um conjunto de passos computacionais a desenvolver para determinar o melhor encaixe entre duas moléculas, sendo elas o receptor e o ligando como está ilustrado graficamente na figura 1.1.

Existem duas vertentes de *docking*: *docking* acoplado (*bound docking*) e *docking* não-acoplado (*unbound docking*). Vakser (2014) [57] indica que o *docking* acoplado é feito com a separação das proteínas de um complexo, voltando a juntar ambas por procedimentos computacionais. Por sua vez no *docking* não-acoplado, também conhecido como *docking* preditivo, o complexo final é obtido por estruturas isoladas. Em termos de computação não existem diferenças, no entanto no *docking* acoplado é mais fácil obter melhores resultados, pois não estão envolvidas alterações de conformação nas estruturas, pelo que estas irão encaixar de forma correta. No entanto a versão não-acoplada é mais utilizada, pois as previsões sobre complexos formados por estruturas isoladas garantem utilidade cientifica.

O problema associado ao *docking* consiste em duas fases: a primeira engloba fazer uma pesquisa sistemática e filtrar as estruturas de proteínas possíveis. A segunda fase consiste em avaliar as possibilidades encontradas na parte anterior de forma a encontrar as corretas [49].

1.2.1 *Docking* entre proteínas

Docking de proteínas por sua vez é definido como a previsão da estrutura tridimensional do complexo de proteínas através das coordenadas atómicas do ligando e do receptor, consistindo nas duas fases anteriormente referidas. O receptor fica estático e ao ligando são aplicadas rotações e translações, sendo determinados os complexos possíveis. O passo final da primeira fase corresponde à filtragem de possibilidades, avaliando estas através de uma função de *score*. Esta função pode avaliar detalhes como a complementaridade de superfície ou de forma através da sobreposição entre os átomos de ambas as proteínas [6]. Como tal a maior parte dos algoritmos de *docking* implementam uma abordagem em que as superfícies moleculares de ambas proteínas são abstraídas em grelhas tridimensionais. Estas grelhas estão divididas por células que permitem guardar a informação associada às regiões internas/externas das proteínas.

Um conceito adicional a ter em conta no momento de filtragem de possibilidades é a rigidez de superfície das proteínas. Existem atualmente três abordagens para tratar a rigidez de superfície das estruturas:

Docking flexível Em que ambos os complexos receptor e ligando são considerados como sendo corpos flexíveis e adaptáveis sendo, no entanto, a mesma flexibilidade interpretada pelo algoritmo de forma simplificada ou limitada e por consequência pode-se aplicar um modelo através de simulações de *docking*.

Docking semi-flexível Um dos elementos do par é considerado rígido e o outro não. Normalmente os algoritmos que implementam o *docking* semi-flexível consideram a superfície molecular do ligando como flexível, devido a ser uma proteína com dimensões menores do que as do receptor, pelo a probabilidade de mudar a sua forma é maior. Outro motivo deve-se aos custos de computação serem mais baixos do que a alternativa de considerar os recetores como flexíveis. Este modelo é considerado apenas em *docking* proteína-ligando, no âmbito de desenho e desenvolvimento de drogas.

Docking rígido A superfície molecular do par é considerada rígida na sua integridade, não sendo possível a sobreposição entre elementos da região central de ambas as estruturas. Adicionalmente considera-se que apesar de ambas as superfícies serem rígidas, uma das proteínas irá acabar por penetrar a outra. Este fator leva a que se tenha de adaptar o conjunto de soluções para o problema em seis dimensões de liberdade: 3 para a rotação e 3 para a translação [57]. Apesar de se considerarem as superfícies de ambos como rígidos, existe uma vertente do *docking* rígido em que é considerada a ocorrência de variações na superfície molecular em ambos ligando e receptor. Esta vertente permite a ocorrência de sobreposições entre átomos de ambas as proteínas desde que seja entre um átomo central e de superfície ou ambos de superfície, tendo o nome *docking* macio (*soft -docking*).

A maior parte dos algoritmos de *docking*, no entanto, adotaram os dois últimos modelos. Desta forma é comum os algoritmos de *docking* entre proteínas seguirem o modelo rígido enquanto que os restantes são adotados por algoritmos dedicados ao *docking* proteína-ligando.

A segunda fase consiste na atribuição de uma pontuação às possibilidades filtradas na fase anterior, através de uma função de *score* com combinações de parâmetros que envolvem contactos residuais, eletrostática até dessolvatação. A gama de parâmetros tem a ver com as características biológicas do par candidato, tendo o objetivo de determinar o quão forte é a interação entre os elementos. Esta fase permite averiguar de que forma o par candidato é correspondido com o par real [3].

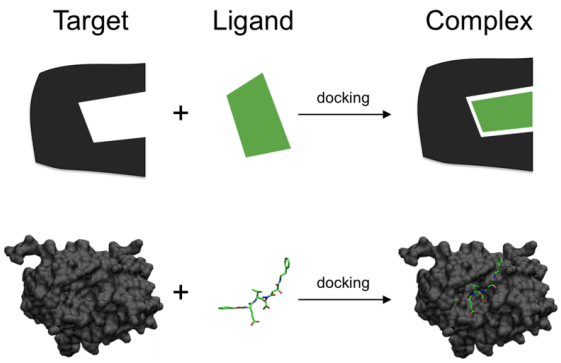

Figura 1.1: Representação gráfica de uma instância de *docking* [46].

1.3 Problema

A GPU (*Graphical Processor Unit*), também conhecida como unidade de processamento de gráficos é uma componente localizada em qualquer placa gráfica instalada num computador. Este dispositivo foi desenhado com o intuito de processar a renderização de gráficos tridimensionais tal como a designação sugere. No entanto a GPU é também adequada para efetuar processamentos alternativos que tal como a renderização de gráficos 3D são demasiado intensos para a CPU abordar por si mesma. O *docking* de proteínas é um exemplo considerável, sendo que a sua execução em GPU apresenta-se como uma solução que permite um aumento considerável de desempenho através da paralelização das suas etapas.

Na atualidade as GPUs oferecem suporte para interfaces de programação e linguagem de alto nível, sendo possível a quem recorra à execução de um programa na GPU alcançar valores de *speedup* superiores em relação a uma implementação para CPU otimizada, em situações em que o problema a resolver envolva computações extensas e repetidas.

O uso da GPU para este tipo de processamentos é referido como *General-purpose computation on graphics processing units* (GPGPU) [12]. Exemplos de aplicações podem variar deste cálculo financeiro até aplicações bioinformáticas, como é o caso do problema de *docking* entre proteínas. É possível encontrar um panorama detalhado sobre as aplicações da computação de alto desempenho sobre forma de catálogo [35].

No âmbito de *docking* entre proteínas e a aplicação da GPU como acelerador de desempenho, existem programas que recorrem a uma adaptação da Transformada rápida de Fourier (FFT) nas etapas iniciais do problema de *docking*. Neste caso uma das opções que os autores adotam para otimizar o desempenho dos seus trabalhos consiste em recorrer a bibliotecas externas especializadas em acelerar as etapas relacionadas com o uso da adaptação , sendo uma das possibilidades a biblioteca cuFFT desenvolvida pela NVIDIA. O uso da cuFFT nestes trabalhos deve-se ao facto de a maior parte do tempo de execução (mais de 70%) de um *docking* simulado num programa que use FFT ser gasto nestas etapas [21, 48], fazendo com o desempenho dos programas dependa da eficiência da biblioteca.

Os programas que usam FFT têm ainda como limitação a largura de cada uma das grelhas tridimensionais necessitar de ser potência de base 2 [41] para serem eficientes. O BiGGER por sua vez determina para as mesmas grelhas uma largura ajustável à largura da molécula, excedendo apenas uma unidade. Deste modo as grelhas do BiGGER têm menor custo em termos de espaço e memória do que as grelhas usadas na adaptação à FFT.

1.4 Solução

A solução consiste em implementar uma nova versão do algoritmo BiGGER, devendo conter otimizações ao código original de forma a que este possa ser executado na GPU. A versão paralelizada do BiGGER deve assumir ganhos de *speedup* consideráveis em relação à versão que apenas utiliza a CPU. A tarefa inicial da implementação da solução passar por identificar as zonas de código do BiGGER que demoram mais tempo e analisar a possibilidade de serem aceleradas por GPU. Na eventualidade de ser possível, definem-se soluções com base em exemplos existentes de forma a otimizar o tempo de execução das zonas identificadas. Sobre a linguagem recorrida para a dissertação, a solução usa as linguagens de programação C++ e Pascal. Como tal a solução recorre à *framework Marrow* destinada à orquestração de computações executadas em OpenCL. Com esta *framework*, para cada otimização implementam-se funções que constroem *kernels* que são submetidos para a GPU, evitando aspetos programáticos de baixo nível que existem quando é usada diretamente a linguagem OpenCL. No entanto como a ferramenta é propriedade do DI/FCT não existe a garantia que as funcionalidades necessárias para implementar a solução existam, pelo que as contribuições para a dissertação englobam a sua implementação e uso.

1.5 Contribuições

A presente tese tem como objetivo contribuir para a introdução de melhorias ao BiGGER, focando nos pontos:

- Identificação de regiões de código do BiGGER que afetam o desempenho deste, podendo ser aceleradas no GPU;
- Adicionar funcionalidades necessárias à *framework Marrow* para implementar a solução;
- Acelerar as regiões problemáticas recorrendo à *Marrow*;

1.6 Estrutura deste documento

O presente documento contém sete capítulos:

Introdução Consiste na introdução sobre os conceitos de *docking* a ter em conta assim como uma formulação do problema computacional associado ao mesmo;

Estado da arte Visão geral em relação aos diversos métodos e ferramentas de *docking*, inclusive o BiGGER, indicando as diferenças em relação a este último. Estas ferramentas surgiram antes de ser considerada a GPU para acelerar a computação associada ao problema. Neste capítulo também está incluída uma secção em que são abordados os conceitos associados à GPU e respetiva programação, assim como o que é que existe em termos de software específico para *docking* com acelerações em GPU relacionado com o BiGGER. Da mesma forma estão presentes os detalhes de implementação e utilidade para as otimizações a desenvolver para o BiGGER. Também são abordados duas APIs para programação em GPU: CUDA e OpenCL;

A *framework Marrow* Breve abordagem à *framework Marrow*, detalhando conceitos essenciais e justificando o uso da *framework* para a solução;

BiGGER versão GPU Procedimentos de implementação da versão do BiGGER acelerada na GPU. É descrito o ciclo de desenvolvimento a seguir que é específico para desenvolvimento de otimizações para execução em GPU, sendo abordadas as etapas do ciclo e os esforços específicos para cada etapa, relativamente à tese. São ainda identificadas as possibilidades de paralelização nas etapas do BiGGER em relação às otimizações apresentadas no capítulo 2;

Contribuições no *Marrow* para suportar a solução final Onde são discutidas as contribuições feitas à *framework Marrow* no âmbito da tese, mais precisamente a adição de novos elementos.

Análise e resultados Ilustram-se os resultados da nova versão do BiGGER desenvolvida no âmbito da tese assim como a discussão dos resultados, comprovando a resolução dos problemas encontrados durante a análise inicial do algoritmo;

Conclusões e trabalho futuro Conclusões sobre o trabalho elaborado e o que pode ser feito após a entrega do presente documento e defesa de tese.

Capítulo 2

Trabalho relacionado

Este capítulo tem o fundamento de apresentar os artefactos de software relacionados com o algoritmo BiGGER [41], sendo que este é o algoritmo abordado com superior relevância por ser aquele sobre o qual vão ser aplicados os esforços para melhorar o desempenho. São esclarecidas as características deste algoritmo e os pontos positivos em relação à concorrência. Os algoritmos e metodologias apresentados na primeira parte do capítulo não têm nenhuma implementação conhecida que permita a sua execução usando o GPU, sendo concorrentes com o BiGGER na versão sequencial.

Tendo em conta o requisito imprescindível sobre a solução final ser executada usando a unidade de processamento gráfico, também conhecida como GPU, são abordados aspetos relacionados com as características deste dispositivo, respetivos modelos de programação e execução assim como boas práticas de implementação de um artefacto de código destinado à execução na GPU.

Na parte final do capítulo apresentam-se três casos de trabalhos relacionados com o da tese, sendo estes casos aplicados aos programas mais conhecidos por profissionais da área. Os pontos essenciais focam-se nas implementações efetuadas para cada caso e de que forma é que podem ser úteis para a solução da tese.

2.1 O algoritmo BiGGER

O BiGGER é um algoritmo para *docking* entre proteínas que considera a superfície destas como rígida, fazendo parte do software para *docking* Chemera. Com a introdução do conceito de *open-source* o programa passou a ter o nome Open-Chemera, sendo este o termo muitas vezes anunciado no decorrer do documento.

Este algoritmo consiste em dois passos: (1) Pesquisa exaustiva e filtragem de possibilidades e (2) Avaliação da interação entre o par de proteínas para cada uma das

possibilidades aceites.

Antes do algoritmo entrar no primeiro passo é efetuada a digitalização da superfície molecular do recetor em grelhas tridimensionais (linha 11 da listagem 2.1). O algoritmo BoGIE usa o conteúdo das grelhas tridimensionais para executar a fase de pesquisa exaustiva e filtragem de poses possíveis (linha 13 e listagem 2.2). Na figura 2.1 está ilustrado o processo de *docking* que o BiGGER segue. No final do primeiro passo apenas existem poucas milhares de possibilidades para a segunda fase avaliar, de um universo com 10^{15} possibilidades.

A segunda fase do algoritmo consiste em aplicar metodologias de aprendizagem automática de modo a que se possa prever qual das configurações resultantes a que tem a interação entre o ligando e o recetor mais forte (linhas 17-21 da listagem 2.1).

Em termos de complexidade temporal, este algoritmo assume ($O(N^{2,8})$) com N a ser o número de nós da grelha, sendo mais rápido do que os algoritmos que recorrem ao *Fast Fourier Transform* [18], também conhecido como FFT. Esta redução do tempo de computação, deve-se à implementação de diversas otimizações face aos algoritmos que usam FFT.

Uma das otimizações é a utilização de uma heurística mais eficiente no passo da eliminação de possibilidades: descarta situações em que existem sobreposições entre átomos centrais do ligando e do recetor ou até mesmo situações que não cumprem com os limites impostos nas restrições introduzidas.

Outra otimização a ter em conta é o tamanho das fatias (matrizes) da grelha, que em relação com as correspondentes em FFT são mais compactas. Para um algoritmo que use FFT ser eficiente, as fatias devem ter uma largura que seja uma potência de base 2, pois a determinação do contacto nas várias posições relativas das fatias é feito pela correlação entre estas. Como tal é necessário que as fatias contenham todas as coordenadas, de forma a que ambas as proteínas possam caber nas matrizes. Este fator pode ser prejudicial em situações em que a soma das larguras das proteínas seja inferior à das fatias sem ser potência de base 2, levando ao uso excessivo de memória. O BiGGER por não depender da FFT supera esta limitação ajustando a largura das fatias à soma referida, tendo em conta que as fatias correspondem apenas ao formato da proteína.

2.1.1 BoGIE

Acrónimo para *Boolean Geometric Interaction Evaluation* [20, 41]. Este algoritmo encontra-se incorporado no BiGGER, assumindo o papel de filtrar as poses candidatas através das superfícies de contacto entre os pares. A filtragem é feita através da avaliação do contacto entre ambas as superfícies dos elementos do par. Na adaptação do pseudocódigo do BoGIE presente na listagem 2.2, é possível verificar que existem dois processos principais a considerar, sendo estes executados para cada candidato. O primeiro consiste na geração de duas grelhas tridimensionais com o intuito de determinar a superfície molecular do ligando (linha 6). No segundo passo ambas as superfícies moleculares do ligando e

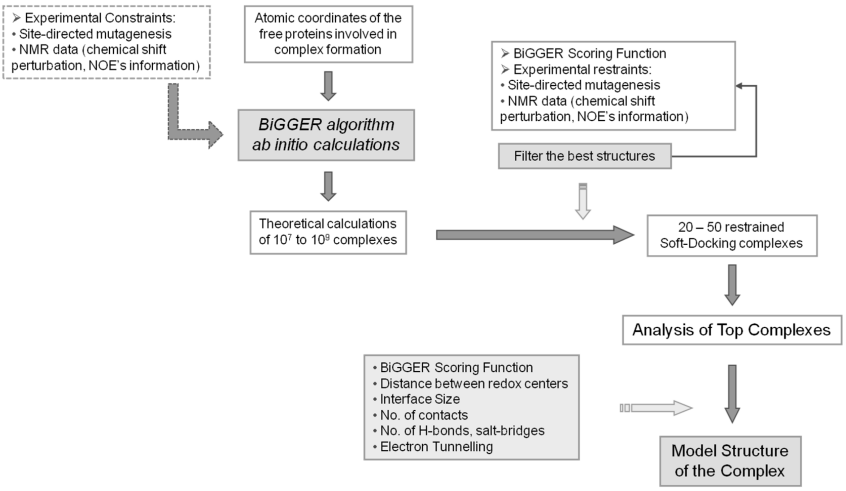

Figura 2.1: Diagrama sobre o processo de *docking* do BIGGER[4].

recetor são confrontadas, avaliando o contacto entre ambas (linha 8). De notar que este algoritmo apenas aborda a primeira fase do problema de *docking*, sendo a fase de *scoring* responsabilidade do BiGGER.

O passo inicial para a determinação da superfície molecular do ligando consiste em construir uma grelha base que define a região interna e externa deste (linha 23). Antes de efetuar a construção o BoGIE determina as dimensões que a grelha necessita para caber na superfície molecular do ligando, resultando em N^3 células equidistantes. Estas células depois são avaliadas em relação à sua pertença á região interna do ligando, atribuindo valores binários (0 ou 1) a estas. O valor 1 é aplicado para todas as células que corresponderem a uma parcela do ligando que se encontra dentro da distância de Van der Waals de um dos átomos constituintes da proteína. Por sua vez é atribuído valor nulo a células que corresponderem à região externa do ligando (linhas 40-44 do pseudocódigo 2.2).

Com a grelha base determinada, o algoritmo procede para a determinação das regiões centrais e de superfície da proteína tal como se encontra ilustrado graficamente na figura 2.2 (linha 24 do pseudocódigo 2.2 assim como as linhas 48-53). A superfície molecular da proteína é determinada deslocando uma cópia da grelha base com a original em 26 possíveis direções. Para cada deslocação é feita uma operação lógica XOR (OU exclusivo) entre ambas as grelhas referidas. Desta forma a operação terá como output o valor 1 apenas nos pontos da fronteira, pois os valores entre as duas células são diferentes e falso se forem iguais. Após esta etapa, o ligando passa a ser representado por duas grelhas central e de superfície.

Com as superfícies moleculares das proteínas determinadas, o BoGIE entra na etapa de filtragem de possibilidades, através da avaliação da sobreposição entre os átomos do recetor e do ligando para cada um destes. A sobreposição entre ambas as proteínas é calculada aplicando uma operação lógica AND entre as superfícies de ambas as proteínas. As possibilidades cuja solução contém sobreposições entre células internas (*core*) em ambas as grelhas são descartadas pois não correspondem a uma solução realista no âmbito

Listagem 2.1: Pseudocódigo BiGGER

```
1
2  iterar e completar os jobs solicitados{
3    // As seguintes estruturas sao actualizadas após a leitura
4    // de um ficheiro xml que descreve o docking
5    float*[3] coordenadasAtomosReceptor;
6    float* distanciasVanDerWaalsReceptor;
7    float*[3] coordenadasAtomosLigando;
8    float* distanciasVanDerWaalsLigando;
9    float*[4] rotacoes;
10
11   digitalizarGrelhas3DReceptor(CoordenadasAtomosReceptor
12     , DistanciasVanDerWaalsReceptor);
13   executarBoGIE(CoordenadasAtomosLigando,
14           DistanciasVanDerWaalsLigando);
15
16   // Fase de scoring: Calcular RSMD da pose usando a sobreposicao mínima
17   iterar todas as possibilidades aceites{
18     iterar constrangimentos introduzidos{
19       verificar se o score nao foi calculado para o candidato corrente
20       se nao foi:
21         calcularRSMDScore(pontos,constrangimento);
22
23       }
24     }
25   }
26 digitalizarGrelhas3DReceptor(float*[3] pontosReceptor
27 ,float* distanciasPontosReceptor){
28     construcaoGrelhaBase( pontosReceptor, distanciasPontosReceptor);
29     construcaoGrelhasSuperficieCentral();
30
31 }
32
33 }
```

do *docking* rígido. No entanto é permitida a sobreposição entre uma célula interna de uma grelha e outra de superfície, favorecendo a atenuação imposta nesta vertente de *docking* (linha 8).

Após o cálculo da sobreposição, o resultado para cada uma das possibilidades é comparada com o pior valor de uma lista com possibilidades aceites e é adicionada a esta se for superior, caso contrário é descartada. Cada elemento desta lista contém o valor para o contacto entre o par de proteínas, um quaternião[9] para a rotação e um vetor tridimensional para a translação. Na eventualidade da lista estar cheia, a possibilidade substitui aquela que previamente estava designada com a pior sobreposição (linhas 10-15). Após a execução do BoGIE existem poucos milhares de possibilidades possíveis para o BiGGER avaliar na fase de *scoring*, de um universo com 10^{15} possibilidades possíveis.

[9]Um quaternião define-se como um número complexo formado por quatro unidades: i, j, k e r. As três primeiras definem um vetor tridimensional que representa o eixo de Euler [59] enquanto que o r representa o ângulo de rotação (em radianos).

Listagem 2.2: Pseudocódigo BoGIE

```
void ExecutarBoGIE(float*[3] CoordenadasAtomosLigando,
    float* DistanciasVanDerWaalsLigando){

             iterar todas as rotacoes e translacoes{
    //Digitalizar a grelha do ligando
    digitalizarGrelhas3DLigando(rotacao corrente);
    //AND entre as grelhas
    int sobreposicao = CalcularSobreposicaoAtomos(GrelhaProbe
    ,GrelhaTarget);
    verificar se a sobreposicao obtida é superior ao pior valor aceite
    se for :
      verificar se a lista encontra-se totalmente preenchida
      se estiver:
        remover pior possibilidade da lista;
      adicionar possibilidade à lista de possibilidades aceites;
    }

}

digitalizarGrelhas3DLigando(float*[3] pontosProbe
,float* distanciasPontosProbe, Quaternion rotacao){
    pontosTmp = rotate(rotacao, pontosProbe);
    construcaoGrelhaBase(pontosTmp, distanciasPontosProbe);
    construcaoGrelhasSuperficieCentral();

}

construcaoGrelhaBase(float*[3] coordAtomos, float* distanciasVdW){
  resolucao = 1.0;
  metade = resolucao / 2;
  iterar todos os pontos da grelha:
    // Conversao das coordenadas dos pontos da grelha de forma
    // a ficar em conformacao com
    // a resolucao da grelha definida pelo BiGGER
    coordenadasDoPontoCorrente = coordenadasDoPontoCorrente
    * resolucao + metade;
    // distância euclidiana entre o ponto e
    // todos os átomos for < distancia de Van der Waals
    // para cada ponto desta
    verificar se o ponto é interno:
    se for
        atribuir valor 1;
    cc
       atribuir valor 0;

}

construcaoGrelhasSuperficieCentral(grelhaBase){

  deslocar uma copia da grelha base nas 26 direcoes adjacentes à original{
    grelhaBase XOR copiaGrelhaBase;

  }
}
```

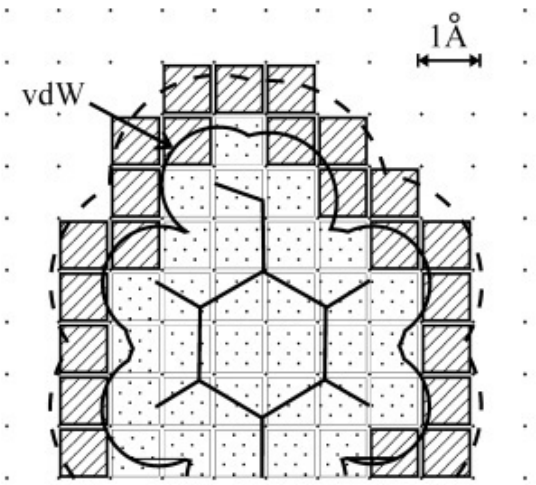

Figura 2.2: Representação em 2D das matrizes resultantes do segundo passo do BoGIE. As células preenchidas a tracejado diagonal correspondem à matriz de superfície, por sua vez as preenchidas com pontos correspondem à matriz *core*. A nuvem de contorno continua representa o corte associado à esfera de van der waals com a proteína localizada ao centro [41].

2.2 A GPU

2.2.1 Arquitectura e Modelo de Execução

Tal como as CPUs, as GPUs também seguem uma arquitetura. Os conceitos essenciais das arquiteturas dos GPUs modernos são transversais aos diferentes modelos existentes, inclusive de diferentes marcas. No presente documento serão utilizadas como referência as arquiteturas dos GPUs Nvidia, em particular a arquitetura mais recente[1] de nome Volta [36] que está presente nos GPUs de modelo Tesla V100 (figura 2.4). Esta nova arquitetura traz diversas otimizações de hardware e lógicas face às versões anteriores, eg. Pascal, Maxwel e Kepler, para desempenho em computações na área do *deep learning*. Também é uma arquitetura própria para acelerações relacionadas com aplicações que usam data-centers.

Em termos gerais as arquiteturas têm os seguintes elementos:

- **Streaming Multiprocessors** : Cada GPU tem uma quantidade variável de *streaming multi-processors* (SMs). Os SMs, por seu lado, são compostos por um conjunto de processadores escalares (SPs) que são também conhecidos como os *cores* da GPU. Os SMs Assumem a função de executar os *kernels* (sobre estes últimos é feita uma descrição detalhada na subsecção 2.2.1.2). Têm frequências de relógio mais baixas, mas suportam paralelismo ao nível de instrução. As componentes dos SMs vão sendo melhoradas face aos SMs de arquiteturas anteriores. A destacar o número de registos que os SMs vão dispondo, a cache L1 e o número de *cores* de execução [60].

[1] O presente documento foi escrito no ano letivo de 2018/2019.

Os SMs são agrupados em partições de hardware com tamanho igual denominados *GPU processing clusters* (GPCs) e o número de GPCs presentes num GPU depende da arquitetura.

Na arquitetura Volta, cada um dos 84 SMs presentes, estão particionados em quatro blocos de processamento como se pode ver na figura 2.3. Cada um destes blocos é composto por 16 cores FP32, 8 cores FP64 e 16 cores INT32. Cada SM é capaz de executar no máximo 2048 *threads*. Foram aplicadas otimizações nos SMs para a versão Volta face a versões anteriores, mais precisamente a adição de *tensor cores*, que são componentes especiais para acelerar as operações associadas a redes neuronais. O GV100 encontra-se dividido em 6 GPCs, cada um destes GPCs contem 14 SMs.

Figura 2.3: Esquema do SM para o GV100[36].

- **Hierarquia de memória**: a GPU contém uma memória global, partilhada por todos os SMs. Esta memória global tem um quantidade de espaço que varia entre 12GB para a arquitetura Kepler e 16GB para a arquitetura Volta. Além disso, existem ainda dois níveis de cache a considerar. As caches L1 são usadas para melhorar a latência das operações globais de escrita e leitura e como especificado no ponto anterior, estas fazem parte dos SMs. Existe ainda uma cache partilhada L2 para complementar a presença das L1. A cache L2 é uma cache de escrita/leitura com uma política de substituição *write-back*. Esta cache responde a pedidos de instruções

load, store assim como instruções atómicas de ambos SM e respetivas caches L1, preenchendo de forma igual as respetivas caches [30]. A partir da arquitetura Volta a cache L1 e a memória partilhada de cada SM passam a estar juntas, o que traz benefícios para a L1 como o aumento da capacidade de memória/SM em 7 vezes a capacidade da arquitetura Pascal, a diminuição da latência de acesso e o aumento da largura de banda [36]. Também existirá uma nova cache de instruções L0 em cada um dos blocos do GV100, melhorando a eficiência face ao uso de buffers de instruções dos SMs anteriores.

Figura 2.4: Esquema da arquitetura da GPU, mais precisamente do Volta GV100[36]. Esta é a arquitetura mais recente que a Nvidia lançou no mercado em 2018/2019.

2.2.1.1 Capacidade de computação de uma arquitetura

Todas as arquiteturas Nvidia têm o conceito de capacidade de computação rotulado a um valor (tabela 2.1). Este valor determina as funcionalidades permitidas pelo hardware respetivo, assim como as melhorias nas componentes de hardware face a arquiteturas anteriores. O número de registos presentes na GPU, o número máximo de *threads* em cada SM e a granularidade de alocação dos registos variam com as diferentes capacidades de computação [32]. O valor da capacidade de computação pode ser usado pelas aplicações em tempo de execução para determinar o que a GPU presente na máquina dispõe em termos de funcionalidades e instruções nativas. Este valor é também decimal, sendo a casa das unidades respetiva ao número de revisão maior e o das décimas ao número de revisão menor. Se uma arquitetura tem um valor de capacidade de computação superior a uma segunda, por ser mais recente, quer dizer a primeira arquitetura tem as funcionalidades e características de hardware da segunda, com mais umas adições novas, assim como a

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Cap. Computação	3.5	5.2	6.0	7.0

Tabela 2.1: Tabela de capacidades computacionais das arquiteturas Nvidia mais recentes [36].

primeira consegue resolver os problemas endereçados na segunda. O que faz com que um programa que tenha sido implementado em referência a uma arquitetura Kepler, com capacidade computacional 3.5, possa ser compatível para execução num GPU com a arquitetura Volta, com capacidade computacional 7.0.

2.2.1.2 Modelo de execução

O modelo de execução em GPU inclui o conceito de computação heterogénea, em que temos dois conjuntos de computações de carácter geral na GPU concorrentes a executar código: o conjunto *host* composto por CPUs e o *device*, composto por GPUs. Os dois sistemas desempenham papeis diferentes. O *host* coordena as transferências de dados a manipular entre as duas entidades e a invocação dos *kernels*. Também gera a alocação de memória nos *devices*. Os *kernels* são funções programadas para executar um determinado número de vezes N em paralelo por N *threads* da GPU, tendo cada uma destas *threads* um ID único, que é acessível dentro do *kernel*. O *device* executa os *kernels* e manipula os dados que o *host* alocou e transmitiu, retornando o resultado para o *host*. As *threads* da GPU são consideradas como *lightweight* pois são escalonadas em grupos conhecidos como *warps*[32][1]. No caso da GPU ter de ficar à espera de um desses grupos, este tem a possibilidade de avançar para outro, pelo que não é necessário haver o sistema de trocas presente na CPU. As *cores* da CPU minimizam a latência para um número muito reduzido de *threads* de cada vez, enquanto que as *cores* da GPU permitem a este gerir um número muito maior de *threads* mais ligeiras, maximizando o *throughput*. Uma computação que é executada pela GPU tem de ser estruturada numa grelha.

Cada um dos elementos da grelha é um bloco referido como *thread block* (na figura 2.5 podemos consultar um exemplo de um *thread block*). A dimensão máxima associada ao tamanho de um *thread block* é dependente da arquitetura. No caso das arquiteturas mais recentes é 1024 *threads*. Numa situação em que é pretendido fazer computações numa estrutura de dados de tamanho superior a 1024 unidades, a mesma é repartida em partes de tamanho igual, sendo cada uma das partes é atribuída a um *thread block* para processamento.

[1] Tendo em conta a quantidade de *threads* individuais que têm de ser geridas e executadas de forma eficiente, é empregado pelos SMs uma arquitetura especifica para o efeito, denominada SIMT (*Single-Instruction Multiple-thread*) [22]. É ainda permitido por parte de qualquer uma das arquiteturas referidas a criação, gestão, escalonamento e execução de *threads* concorrentes em grupos de 32 cada. Estes grupos denominam-se *warps*, podendo cada bloco de *threads* do CUDA ter 1 ou mais warps [30].

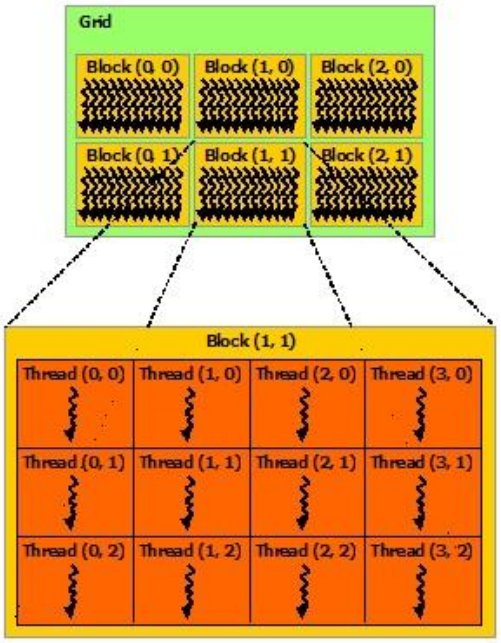

Figura 2.5: Ilustração de uma grelha de *thread blocks*, detalhando um elemento da grelha para ilustrar os pormenores de um bloco de *threads*. Neste caso o bloco é bi-dimensional, mas existe a possibilidade de ser tridimensional, o mesmo se aplica à grelha [33].

2.2.2 Modelos Base de Programação

Em termos de programação em GPUs existem como APIs o CUDA (*Compute Unified Device Architecture*)[34] que foi implementado pela Nvidia e o OpenCL (*Open computing language*)[37]. Ambos suportam a linguagem C/C++ apesar de poderem suportar outras linguagens, como por exemplo Pascal no caso do OpenCL. O CUDA apenas funciona com placas da Nvidia enquanto que o OpenCL permite efetuar paralelizações em hardware de diferentes arquiteturas e tipos, desde CPUs a *clusters*. Sobre o CUDA existem *bindings*[3] para outras linguagens, como por exemplo pyCUDA para a linguagem Python ou a biblioteca especializada com o cuFFT [38] para acelerar a técnica FFT. Por sua vez o modelo de programação em OpenCL oferece uma linguagem comum, interfaces de programação e abstrações em hardware, permitindo ao programador acelerar aplicações com computações paralelizadas por dados ou por tarefas, num ambiente de computação heterogénea [53].

O programador deve elaborar código em duas vertentes. Por um lado tem de programar as tarefas do lado do *host*, mais precisamente alocações de memória na GPU, eventuais transferências de dados entre a CPU e a GPU que vão ser manipulados na execução dos

[3] No contexto da informática uma *binding* diz respeito a uma integração da linguagem em outras linguagens.

kernels e quando é que estes são invocados. Sobre os *kernels* o programador tem ainda de especificar os parâmetros de execução (dimensão do bloco e número de *threads* a invocar no *kernel*), sendo que com estes parâmetros o CUDA/OpenCL determina a quantidade de *threads* a lançar. Por outro tem de implementar o código que cada *thread* da grelha deve executar, a correr dentro do *kernel* em que pretende fazer a respetiva paralelização. O código sequencial deve ser executado no *host* e o código a paralelizar no programa deve ser executado no *device*.

2.2.3 Optimizações

Um dos passos na programação de versões aceleradas em GPU para um programa consiste em aplicar um conjunto de possíveis otimizações à versão paralelizada do programa, de forma a otimizar a performance do programa para que se possa comparar o resultado final com as expetativas iniciais. Existem quatros aspetos a considerar quando pretendemos otimizar um programa recorrendo à GPU, com os respetivos detalhes de acordo com o guia de programação em CUDA [32]:

- **Sobreposição de comunicação com computação** : É necessário sobrepor as computações do *host* e do *device* com a comunicação pois ter as duas sem sobreposição pode afetar o desempenho de um programa paralelizado. Em CUDA tal pode ser feito através de *streams*. *Streams* são sequências de operações que são executadas no *device*, por uma dada ordem imposta pelo *host* podendo ser cópias de memória ou execuções de *kernels*. Apesar das operações numa *stream* terem de ser executadas pela ordem imposta pelo *host*, as operações entre *streams* podem ser interligadas, havendo sobreposição, e por consequência, possível a esconder a latência associada à transferência de dados entre o *host* e o *device*. Dependendo da arquitetura do *device*, é possível sobrepor a comunicação entre *host* e *device* e a computação respetiva à execução do *kernels*. O requisito é ambas as instruções serem de *streams* diferentes e não por omissão[2], caso contrário as instruções terão de ficar à espera de instruções anteriores no *device* terem acabado, sem poderem começar, o que impossibilita a sobreposição das instruções e esconder a latência de comunicação.

- **Taxa de ocupação da GPU** : É essencial, para o desempenho ser ótimo, manter os SMs da GPU o mais ocupados possível no decorrer da execução da aplicação, devendo existir uma distribuição de trabalho equilibrada entre os SMs. A aplicação final deve estar implementada de forma a que use os *threads* e respetivos blocos maximizando a utilização do hardware, evitando situações em que se deixa de impor a distribuição livre de trabalho entre os SMs. A taxa de ocupação determina o quão efetivamente a GPU se encontra ocupada, isto é, o número de *warps* que estão ativos em relação ao número máximo de *warps* que a GPU consegue ativar. É

[2] Uma *stream* por omissão tem o seu *streamID* com o valor nulo. Deste modo o pretendido é uma *stream* cujo o seu id seja diferente de 0.

pretendido que esteja o mais próximo possível de um certo limite que depende da capacidade de computação da arquitetura da GPU [3]. Exceder este limite não traz melhorias de performance, no entanto se o código estiver longe de atingir este limite é garantido que o desempenho não vai ser ótimo. Para garantir a taxa de ocupação adequada à GPU, é possível optar por garantir que os *kernels* são executados ao mesmo tempo, o que é chamado de execução concorrente de *kernels*.

- **Optimizações de memória** : Em que são consideradas as memórias global e partilhada do *device*. Sobre a memória global, esta é acessível via transações de memória de 32, 64 ou 128 bytes. Estas transações devem estar naturalmente alinhadas, o que implica apenas os segmentos de memória cujo primeiro endereço é um múltiplo do tamanho do segmento, sendo este 32, 64 ou 128 bytes poderem ser escritos ou lidos pelas transações. Quando um *warp* executa uma instrução que pretende aceder à memória global, é feita a coalescência dos acessos à memoria por parte das *threads* do *warp* numa quantidade de transações de memória que depende do tamanho da palavra acedida por cada *thread* e da distribuição dos endereços de memória pelas *threads* do *warp*. Quanto maior é o número de transações necessárias, maior é o número de palavras não usadas que são transferidos em adição às palavras acedidas pelas *threads*, o que tem como efeito a redução do *throughput* de instruções[33].

  No caso da memória partilhada, esta tem uma latência de acessos menor do que a memória global e largura de banda superior. A memória partilhada está dividida em módulos de memória com tamanho igual chamados *banks*. Os *banks* podem ser acedidos de forma simultânea. Existe a possibilidade de ocorrer *bank conflicts* quando dois endereços de um pedido de acesso à memória correspondem ao mesmo *bank*. Deste modo o acesso deve ser serializado e o pedido dividido em sub-pedidos separados que são livres de conflito. Por consequência o *throughput* é reduzido em um fator que depende do número de divisões efetuadas.

- **Controlo de fluxos** : É muito importante evitar que ocorram divergências na execução de *threads* de um mesmo *warp*. Esta situação pode acontecer quando dentro do código de um *kernel* existem instruções de controlo de fluxos (eg. if, switch,while, do while e for) o que leva à redução do *throughput* de instruções devido ao facto de existirem *threads* dentro de um *warp* a divergir em caminhos de execução diferentes. No entanto podem existir situações em que o fluxo de controlo depende unicamente do *thread* ID, nessas situações é importante a escrita da condição de controlo de forma a atenuar o número de *warps* a divergir. Outra forma de garantir que não existem divergências é tornar fácil para o compilador [4] o uso de *branch predication* ou seja, o compilador desenrola os loops/condições impedindo divergências de *warps*.

[3] Como foi descrito no ponto 2.2.1, a respeito da capacidade de computação de uma arquitetura, o respetivo valor está associado ao número de registos presentes em cada SM, que pode variar dependendo do valor da capacidade.

[4] No caso do CUDA o compilador é o NVCC, no caso do OpenCL é o OpenCL Compiler.

Apenas as instruções em que o predicado assume o valor verdadeiro em relação à condição de controlo são executadas.

2.3 *Docking* em GPU

Existem vários trabalhos que focam a paralelização de etapas constituintes do processo de *docking* na GPU[5]. Destes, são abordados nesta secção o Megadock [39, 48], o PIPER [21, 54] e o AutoDock [17]. O Megadock e o PIPER são os programas mais conhecidos cujo funcionamento é semelhante ao do BiGGER. No caso do Megadock, este é um dos programas em que o seu desempenho aumentou drasticamente após a implementação da versão 4.0 que suporta aceleração na GPU. O PIPER, num contexto histórico, foi um dos primeiros programas para *docking* a ser acelerado usando o CUDA. O caso de estudo mostra ainda a importância de efetuar a manutenção do código acelerado de versões anteriores e o seu impacto no desempenho do PIPER quando é executado com hardware mais recente. O AutoDock, apesar de ser um programa cujo funcionamento é diferente do BiGGER, foi considerado pois foi desenvolvida uma paralelização à etapa de *scoring* onde são discutidas duas abordagens em função da taxa de ocupação da GPU assim como é discutida a possibilidade sobre o uso da memória partilhada deste. Ambos Megadock e PIPER são programas que utilizam a Transformada rápida de Fourier, também conhecida como FFT [18]. Esta é a abordagem padrão dos programas de *docking* rígido entre proteínas, sendo o BiGGER e o BoGIE formados a partir desta técnica [41]. Deste modo considerou-se necessário abordar esta técnica para além dos trabalhos de paralelização.

2.3.1 Transformada Rápida de Fourier

Antes da introdução do conceito de GPGPU, uma das formas de efetuar o passo de reconhecimento da superfície molecular para cada candidato numa pesquisa exaustiva seria aplicar correlações de espaço real. Estas correlações são efetuadas num espaço de 6 dimensões, pelo que a complexidade temporal é assumida como $O(N^6)$ [20], sendo N o número de nós das grelhas de correlação usadas no processo. Katchalski Katzir et al [18] abordaram em 1992 uma adaptação da Transformada de Fourier que permite efetuar o mesmo passo com uma redução da complexidade temporal para $O(N^3 * log_2(N^3))$, sendo N o número de nós presentes na grelha.

A metodologia com que são determinadas as possibilidades consiste, de forma resumida, nos passos:

Reconhecimento geométrico Neste passo inicial o algoritmo abstrai as superfícies moleculares de ambos ligando e recetor numa grelha tridimensional, recebendo as coordenadas atómicas de ambos assim como os respetivos raios de Van der Waals (VdW) como input. A grelha tridimensional encontra-se dividida em células que

[5] No âmbito de investigação de casos de estudo, foram encontrados favoráveis os programas Megadock, PIPER, HEX, AutoDock e o MolDock. Estes dois últimos são para *docking* apenas entre proteína-ligando.

representam uma dada região molecular e serão submetidas a uma avaliação que determina se a região é interna ou externa. Cada célula é considerada como interna se estiver dentro do raio de VdW de pelo menos um átomo e externa em caso contrário.

Após a definição das regiões interna e externa a região de fronteira é determinada para ambas as proteínas. O método considera uma camada superficial fina como a região de superfície, mantendo para as células correspondentes o valor 1. São determinadas duas funções $\bar{a}$ e $\bar{b}$ que permitem suportar os pontos de fronteira. Para a função $\bar{a}$, cada célula da grelha pode ter como valores atribuídos: 1 para correspondência a coordenadas atómicas localizadas na fronteira, ρ a coordenadas internas e 0 a externas. O procedimento é idêntico para a função $\bar{b}$, apenas substituindo ρ por δ.

Correspondência de superfícies Após o reconhecimento geométrico ter sido efetuado, procede-se para uma etapa em que é simulado o contacto entre ambas as superfícies moleculares. O contacto é avaliado determinando uma função $\bar{c}$ que recebe como argumento um vetor de deslocação. Se para um dado vetor de deslocação se verificar que não existiu contacto a função retorna valor nulo. Este valor pode ser positivo no caso de existir contacto sem penetração e negativo se for verificado que o ligando penetrou o recetor. Para diferenciar entre contacto e penetração, os valores das células que previamente foram classificadas como internas devem ser excessivamente negativos no caso do recetor e positivos mas próximos de 0 no caso do ligando. Deste modo é garantido que a correlação retorna valores positivos quando é verificado um contacto sem penetrações.

A determinação direta da função de correlação $\bar{c}$ usando diretamente $\bar{a}$ e $\bar{b}$ seria um passo computacionalmente pesado pois são necessárias N^6 operações de soma e multiplicação, sendo N o número de nós na grelha. Em alternativa os autores do método optaram pela aplicação da Transformada de Fourier que garante ser uma alternativa mais rápida. São calculadas as Transformadas discretas de Fourier (DFTs) [9] das funções $\bar{a}$ e $\bar{b}$, tendo como resultado duas funções $A*$ e B. O produto destas funções resulta numa função C. A função $\bar{c}$ é determinada por C através do cálculo da Transformada Inversa de Fourier (IFT) [9], sendo estes cálculos efetuados para todas as deslocações possíveis do ligando com o recetor.

Avaliação do contacto Os valores de $\bar{c}$ são analisados. Um exemplo gráfico do conjunto de resultados ilustra-se na figura 2.6. É possível observar que em condições favoráveis existirá uma instância do vetor de deslocamentos para o qual o valor da IFT será máximo. Esta instância representa um possível complexo, pelo que o algoritmo a deve guardar para processamento posterior.

Estes passos são iterados para cada uma das possíveis orientações que o ligando pode assumir em relação ao recetor. Sobre um eixo xyz os ângulos que a orientação do ligando pode formar variam entre $360x360x180\Delta^3$, sendo Δ o intervalo de amostragem rotacional.

O algoritmo termina com uma ordenação decrescente da altura para todos os picos encontrados, sendo os complexos correspondentes às melhores possibilidades encontradas.

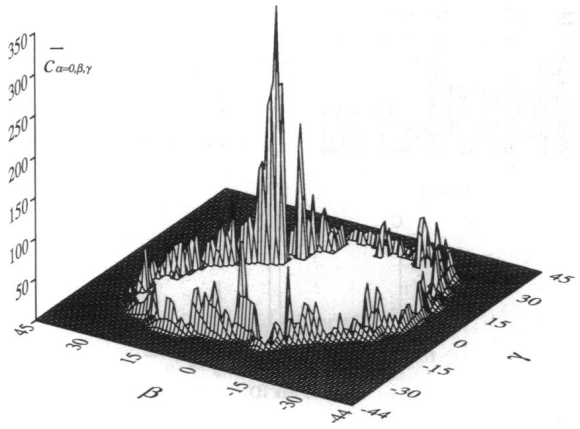

Figura 2.6: Exemplo de uma secção transversal com o resultado gráfico dos valores da função de correlação de C para os vetores de deslocamento [18].

2.3.2 Megadock

O Megadock 4.0 [39] é um software de protein-protein *docking* de origem japonesa atualmente com suporte *CPU-only*, *single GPU* e *multi-GPU*. Este programa usa a técnica de *Fast Fourier Transform* na sua correlação de matrizes.

O facto de suportar execução multi-GPU garante a adequação do programa para executar em máquinas que têm muitos *cores* de GPU e CPU à disposição como por exemplo supercomputadores e clusters de computação. No entanto é possível utilizar o Megadock em computadores pessoais, fazendo as alterações respetivas no seu ficheiro de compilação. O funcionamento do Megadock 4.0 com execução multi-GPU envolve a criação de um processo *master* que faz a aquisição de uma lista de pares de proteínas e distribui o *docking* dos pares para os workers presentes nos nós disponíveis. Estes, por sua vez, distribuem o trabalho de calcular a rotação do ligando em cada nó da lista, pelos diversos GPUs e CPUs do nó do *cluster*. A execução pelos GPUs de cada nó é feita por CUDA e pelos CPUs por OpenMP. Uma das vantagens que este protocolo assume é a tolerância a falhas pois o nó *master* consegue supervisionar os resultados dos *jobs* executados pelos workers, além disso é escalável com o número de elementos que compõem o *cluster*.

2.3.2.1 Execução em GPUs

As acelerações implementadas para o Megadock consistem em otimizações para 6 etapas constituintes do programa. Na figura 2.7 ilustram-se as etapas no processo de *docking* no Megadock, foi aplicado um *profile* sobre o funcionamento do programa com apenas 1 core da CPU, registando os tempos de execução em cada etapa. Os autores do trabalho concluíram que todas as etapas entre P4 e P8 consomem a maioria do tempo de execução. Estas etapas constituem um ciclo em que se iteram todas as rotações possíveis do ligando

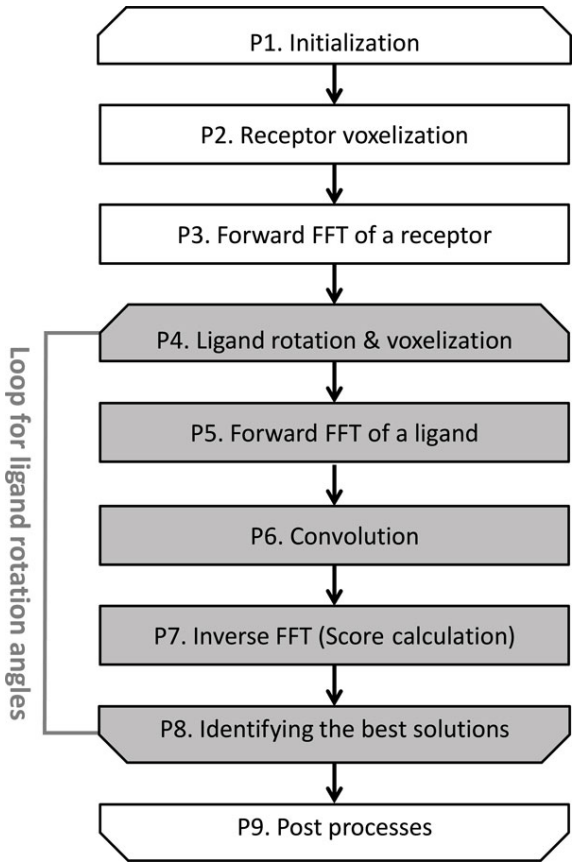

Figura 2.7: Etapas do processo de *docking* no Megadock. As etapas a cinzento foram aceleradas por GPU[48].

em relação ao recetor. No caso da etapa P4 as coordenadas do ligando são atualizadas de acordo com uma dada matriz de rotação e o processo respetivo é independente para cada átomo, sendo paralelizável. Esta etapa foi acelerada mapeando as coordenadas atómicas do ligando para a GPU. A segunda vertente da P4 consiste na voxelização do ligando, em que é feita uma afetação a uma posição da grelha em relação às coordenadas atómicas de um dado átomo do ligando. As coordenadas devem pertencer à região interna da curva de van der Waals. Este processo é também paralelizável em relação a cada átomo. Nesta vertente os átomos também são processados em paralelo e mapeados para a GPU, sendo cada átomo designado a uma *core* da GPU.

Para as etapas P5 e P7, em que é feita a FFT discreta do ligando e o cálculo da inversa do FFT deste, respetivamente, foi utilizada a biblioteca especializada para FFT, cuFFT. Na etapa P6 denominada convolução foi aplicado um mapeamento para a GPU. Na etapa P8 os resultados têm uma dada pontuação, sendo a etapa paralelizada aplicando uma operação de redução entre a coleção de resultados. O pretendido com a redução foi determinar o resultado com o *score* ótimo. Sobre a comunicação *host-device*, é referido que a transferência de grandes porções de dados é única em toda a execução do programa. O conteúdo da transferência inclui as coordenadas atómicas do ligando e a grelha do recetor com o FFT aplicado. No entanto para cada uma das rotações a iterar ocorre a transferência

dos dados necessários para determinar as coordenadas dos átomos do ligando em função do ângulo de rotação.

Em termos de performance e tempos de execução, os testes efetuados em 2014 revelam que a implementação MIC do Megadock 4.0 foi capaz de fazer em 3H, usando 420 nós, um caso de teste que requer 250.000 *dockings* que versões anteriores do Megadock levariam dias [39]. A versão acelerada na GPU demonstrou ser 15.1 vezes mais rápida do que a versão sequencial.

2.3.3 PIPER

À semelhança do Megadock, o PIPER é um programa de *docking* entre proteínas baseado em grelhas FFT para calcular a complementaridade de superfície. A implementação do PIPER trouxe a introdução do uso da energia de dessolvatação do par na função que avalia os complexos possíveis, complementando as funções de *score* respetivas à forma do complexo e a eletrostática. O fluxo de programação do PIPER consiste em duas fases: a fase *setup* envolve a leitura dos dados relacionados com as moléculas, que são passados como input em ficheiro, a computação dos passos relacionados com o recetor, e a criação das grelhas do ligando por correlação em FFT. Após o *setup* estar concluído são iteradas as possíveis rotações, em que as etapas referidas na figura 2.10 são aplicadas para cada rotação.

A implementação de acelerações ao PIPER [54] por GPU ocorreu inicialmente em 2009. No entanto em 2014, a performance da versão PIPER GPU de 2009[6] foi confrontada com a versão CPU de 2014[7] através da execução de um *profile* à performance das duas versões. A versão CPU2014 introduziu otimizações no algoritmo original, mais precisamente foi alterada a biblioteca que aplicava o FFT. Verificou-se que a versão CPU2014 conseguiu ter performance superior às acelerações introduzidas em 2009, com execuções utilizando um GPU de 2014 [21]. A proporção de tempo gasto no passo de correlação das matrizes é maior na versão CPU do que na GPU09 como se pode observar na figura 2.11. No entanto, as proporções para a filtragem, acumulação e cálculo do *score* assim como a atribuição de grelha e rotações é maior na versão que usa GPU do que na versão CPU. Deste modo foi desenvolvida em 2014 uma solução com acelerações em GPU que aborda a paralelização dos passos de filtragem, atribuição de grelhas e transferência de dados entre o *host* e o *device*[8]. Ambas as soluções foram desenvolvidas em CUDA.

Sobre o passo de filtragem e cálculo do *score*, para a versão de 2009 foi implementado um *kernel* para a filtragem e atribuição de *score* para cada um dos conjuntos de coeficientes que podem ser adicionados à função que determina a energia do par, para uma rotação. O caso de uso ideal consistia em utilizar 8 desses conjuntos ao mesmo tempo e utilizar 1

[6] Esta versão será referida ao longo do texto como GPU09
[7] Esta versão será referida ao longo do texto como CPU2014
[8] Esta versão será referida ao longo do texto como GPU2014

SM para calcular o *score* máximo de cada um dos conjuntos[9]. Esta otimização deixou de ser válida pois em 2014 uma das otimizações na versão CPU2014 do PIPER foi reduzir o número total de conjuntos de coeficientes a ser processados sendo que na versão GPU09 apenas um SM da GPU estava a ser utilizado. Passou a ser pretendido para o passo de filtragem encontrar o *score* ótimo para um conjunto de coeficientes no menor tempo possível, repetindo o mesmo procedimento para cada um dos restantes conjuntos.

A versão GPU2014 introduziu nestes dois passos a adição de dois *kernels* em alternativa a usar apenas um na versão GPU09 para os dois passos. Ambos os *kernels* partem o trabalho total em duas fases de forma a que a memória partilhada da GPU possa ser utilizada para acessos rápidos de memória assim como o trabalho possa ser distribuído por todos os SMs e são repetidos para cada um dos conjuntos.

O primeiro *kernel* particiona por todos os SMs desocupados os dados da grelha molecular (figura 2.8). Este *kernel* é lançado com um número de *thread blocks* que permita que cada SM fique ocupado com uma quantidade adequada de trabalho. Em cada um dos *thread blocks*, cada *thread* acede a uma parcela do output, calcula o *score* ótimo dentro do subconjunto e guarda o resultado num endereço de memória partilhada para o bloco correspondente. O acesso ao subconjunto segue as características apontadas na subsecção 2.2.3 em relação às otimizações de memória. O *kernel* é finalizado quando cada *thread* executada determina o *score* ótimo geral para cada bloco e guarda este na memória global.

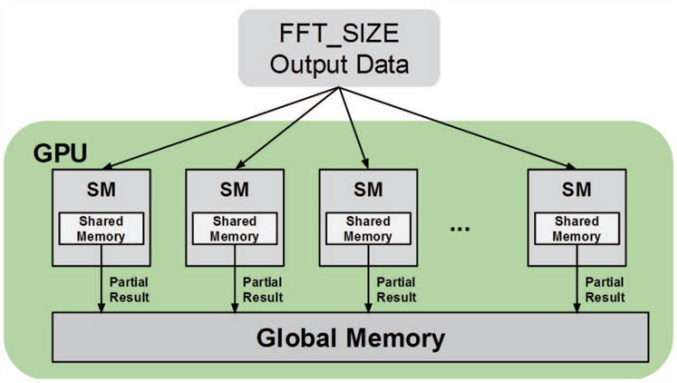

Figura 2.8: Esquema para o primeiro *kernel* introduzido na versão GPU2014 [21].

O *kernel* anterior usa mais do que um *thread block*, por sua vez o segundo *kernel* apenas usa um bloco de *threads* (figura 2.9). O bloco a usar tem de ter o mesmo número de *threads* que o número de blocos que foram usados no passo anterior. Cada uma das *threads* do bloco compara dois *scores* e escreve o melhor dos dois na memória partilhada para o bloco, sendo feita uma sincronização para garantir que as *threads* operam no mesmo passo de iteração e sobre memória consistente. O *score* ótimo é determinado e guardado na memória global.

[9] A solução GPU09 foi desenhada com a GPU Tesla C1060 em mente, esta arquitetura tem 30 SMs, ficando com 8 ocupados na execução do *kernel*, o que garante uma performance melhor do que uma versão do PIPER que apenas recorre à CPU.

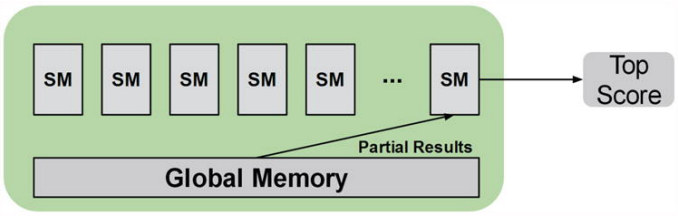

Figura 2.9: Esquema para o segundo *kernel* introduzido na versão GPU2014 [21].

Para além da adição dos dois *kernels* referidos anteriormente, o tempo em que a GPU não desempenha funções enquanto que a rotação e atribuições na grelha estão a ser processados pela CPU foi eliminado. A computação da grelha do ligando passa a ser desempenhada pela GPU, deixando de existir a necessidade de a transferir entre as memórias do *host* e da GPU. Esta otimização tem como requisito um GPU com memória global suficiente para armazenar os *arrays* relacionados com as atribuições da grelha do ligando.

No entanto houve otimizações da solução GPU09 que foram mantidas. Nas etapas em que a aplicação do FFT às grelhas é feita, a otimização original consistiu em aplicar a biblioteca cuFFT à semelhança do que foi feito para o Megadock sobre os passos que envolvem FFT sobre as grelhas. As restantes etapas foram paralelizadas mapeando todo o processo para a GPU, novamente à semelhança da implementação para o Megadock.

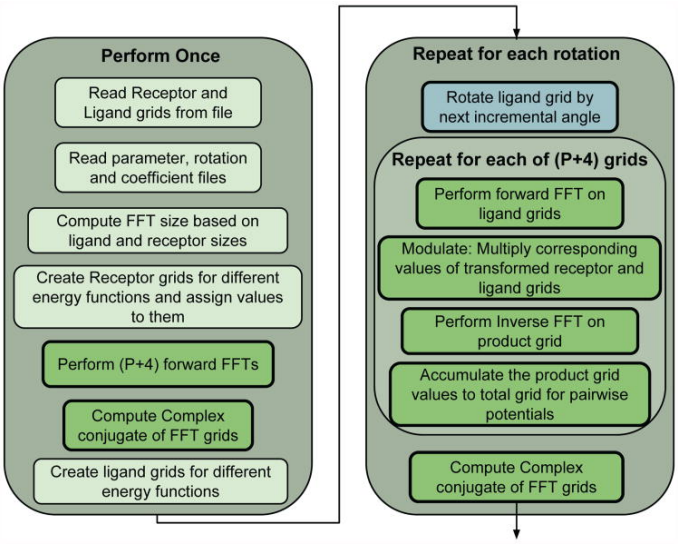

Figura 2.10: Etapas do processo de *docking* no PIPER. As etapas destacadas a verde escuro foram aceleradas por GPU em 2009 e a etapa a azul em 2014 [21].

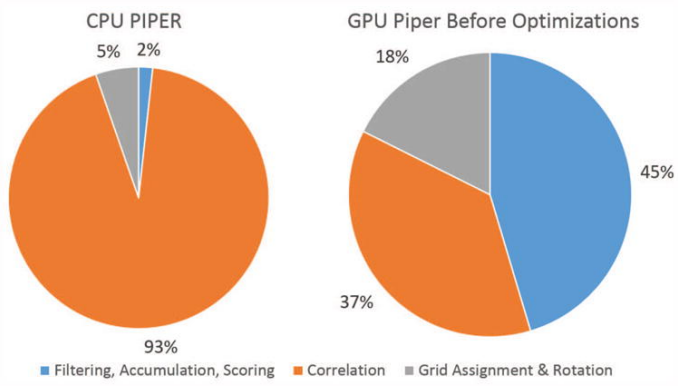

Figura 2.11: Proporções de tempo gasto na execução para a versão CPU otimizada e a versão de 2009 que usa GPUs [21] .

2.3.4 AutoDock

O AutoDock [28] é um programa diferente dos até agora indicados, não só por ser específico para interações entre proteína-droga[10] como também por utilizar o algoritmo genético de Lamarck [29] para determinar a posição correta dos elementos do par, em alternativa ao uso da complementaridade de superfície como função de *score*. Deste modo o AutoDock descarta as operações binárias que o BiGGER usa e as correlações usando FFT que os outros programas usam. atualmente, existem duas versões do software: AutoDock 4 e Vina [55]. O AutoDock 4 está dividido ainda em dois sub programas: *autodock* executa o *docking* do ligando com um conjunto de grelhas que fazem a descrição do complexo resultante. O segundo, *autogrid*, faz os cálculos prévios com o objetivo de obter as grelhas que o AutoDock necessita para desempenhar as suas funções. O AutoDock Vina é diferente do AutoDock 4 no sentido de não efetuar o cálculo das grelhas de forma prévia mas sim instantaneamente, de forma automática, não guardando as grelhas em disco.

Em 2010 foi abordada, à semelhança do que foi feito para o Megadock assim como para o PIPER, uma possível paralelização do AutoDock utilizando a GPU [17]. Neste caso a API utilizada foi o CUDA, sendo desenvolvida a versão 4.2.1 do AutoDock que permite um aumento de *speedup* em 2x, sendo este aumento semelhante ao registado para o Vina sobre o AudoDock 4.

Na determinação de uma abordagem para paralelizar o AutoDock, foram consideradas duas opções. A primeira consistia em dedicar cada *thread* da GPU ao cálculo da energia total de um individuo, sendo esta opção de nome *PerThread*. A segunda opção de nome *PerBlock* consiste em dedicar um bloco de *threads* da GPU para determinar essa energia total. A primeira solução garante a saturação da GPU (taxa de ocupação alta) para milhares de indivíduos e a segunda oferece a mesma saturação para centenas de indivíduos. A solução optada foi uma variante do *PerBlock*, em que existem 5 *kernels* de escrita/leitura para cada passo do algoritmo genético (inicialização de coordenadas atómicas do individuo, aplicar a torção até ao cálculo da energia interna) para as coordenadas atómicas

[10] Uma droga é caracterizada por uma molécula de pequenas dimensões.

respetivas a um individuo. De forma a eliminar a possibilidade de existir *overhead*, foi optado lançar um *kernel* e guardar as coordenadas atómicas na memória partilhada. Por este fator, a solução foi renomeada para *PerBlockCached*.

2.4 Sumário

Neste capítulo foi efetuada uma análise aos algoritmos BiGGER e BoGIE, destacando as características que este par algoritmos assume e vantagens em relação aos algoritmos concorrentes. Tendo em conta que o tema da dissertação envolve a otimização em GPU deste par de algoritmos, foi dado ênfase à GPU assim como à programação em CUDA e OpenCL. Foram abordados os conceitos de programação que são necessários ter em conta quando pretendemos otimizar um projeto com tarefas embaraçosamente paralelas, recorrendo à GPU para as executar.

Adicionalmente foram estudados três casos de trabalhos que revelaram ser semelhantes no contexto da corrente tese. Sobre a implementação GPU do Megadock, em que foi usado CUDA, existem aspetos que podem ser aproveitados para ajudar na paralelização do BiGGER. Os mecanismos *master-worker* não são tão importantes pois o trabalho a desenvolver na elaboração da dissertação não engloba a execução multi-GPU do BiGGER. As abstrações usadas na implementação dos *kernels*, no entanto, podem ser aproveitadas. Adicionalmente este caso mostra que o mapeamento das etapas de *docking* para a GPU tem custos de implementação. Os autores do Megadock referem que no total foram acrescentadas 1000 linhas de código às 7000 que o programa tinha originalmente, considerando o custo como elevado. As 1000 linhas adicionais incluem não só *kernels* como mecanismos que facilitam a transferência de dados entre o *host* e o *device*.

O caso de estudo relacionado com o PIPER mostra que nem sempre uma versão de um programa com acelerações em GPU é superior a uma outra versão mais recente do mesmo programa que use a CPU com otimizações. Por consequência é necessário adaptar o código às funcionalidades que as arquiteturas GPU correntes suportam assim como às alterações que o programa original sofre. As otimizações que foram aplicadas ao PIPER em 2014 foram aplicadas à fase de *scoring*, mais precisamente às etapas de filtragem e cálculo dos *scores* considerados como ótimos. É mostrada uma forma de otimizar a fase de *scoring* do BiGGER, implementando uma redução como foi confirmado no caso do Megadock para a mesma etapa.

Sobre o caso do AutoDock, foi demonstrado na subsecção 2.3.2 uma possibilidade de solução para acelerar o BiGGER, em que é implementado um conjunto de *kernels* para as diversas etapas, inclusive uma redução para determinar a solução ótima e mapeamentos de dados para a GPU. A solução para o AutoDock oferece uma forma de poder acelerar o cálculo da função de *score* do BiGGER utilizando a memória partilhada da GPU.

No capítulo seguinte será discutida a *framework Marrow*, componente essencial para implementar a solução proposta na tese.

Capítulo 3

A *framework Marrow*

Deste capítulo para a frente serão abordados os detalhes relativos à implementação da versão paralelizada do BiGGER, sendo este capítulo dedicado à biblioteca de esqueletos algorítmicos *Marrow* [2, 24, 25, 50].

Esta biblioteca destina-se à orquestração e execução de computações baseadas em OpenCL no GPU ou num conjunto destes. É a primeira biblioteca da espécie a suportar esqueletos para paralelismo de tarefas assim como o aninhamento de esqueletos em árvores de computação(*Skeleton Computational Trees*) [25]. De notar a importância do conceito de esqueleto (*skeleton*) que é um modelo de paralelismo generalizado, podendo ser criados modelos mais complexos a partir dele. Os esqueletos têm um nível da abstração alto, permitindo a abstração de complexidades apresentadas em programas paralelizados. Adicionalmente podem encarregar-se, para além do paralelismo, da execução das etapas de sincronização e de comunicação. A incorporação do *Marrow* na solução para a tese permite ao BiGGER execução multi-GPU e em diferentes dispositivos de marcas que não a Nvidia, devido ao fator de abstrair para OpenCL e não para CUDA. Adicionalmente permite implementar as soluções a um nível superior de abstração em C++ e não diretamente em OpenCL. Este ponto é positivo pois garante manutenção fácil de código assim como eficiência em termos de número de linhas. O *Marrow* no entanto por ser uma ferramenta atualmente a ser desenvolvida pelo DI-FCT/UNL não garante a presença dos elementos ideais para uma dada solução, ficando ao encargo do desenvolvedor a implementação destes.

3.1 Arquitetura

De acordo com o artigo de Fábio Soldado et al (2016) [50] referente ao *Marrow*, a arquitetura deste divide-se principalmente em três componentes (figura 3.1). A camada

biblioteca de esqueletos (*skeleton library*) contém as entidades que o desenvolvedor pode recorrer para implementar soluções. Este nível contém entidades como os próprios *skeletons* e estruturas de dados (*containers*). A camada *runtime* diz respeito à compilação e execução dos *kernels*, dando importância a aspetos como alocação e transferência de memória para o GPU. Por fim existe a camada respetiva ao dispositivo OpenCL, onde são considerados os aspetos relacionados com o dispositivo e respetiva plataforma de execução. Cada camada apenas tem acesso às componentes das camadas adjacentes. Existe ainda a camada das aplicações C++, estando esta camada um nível acima da biblioteca de esqueletos.

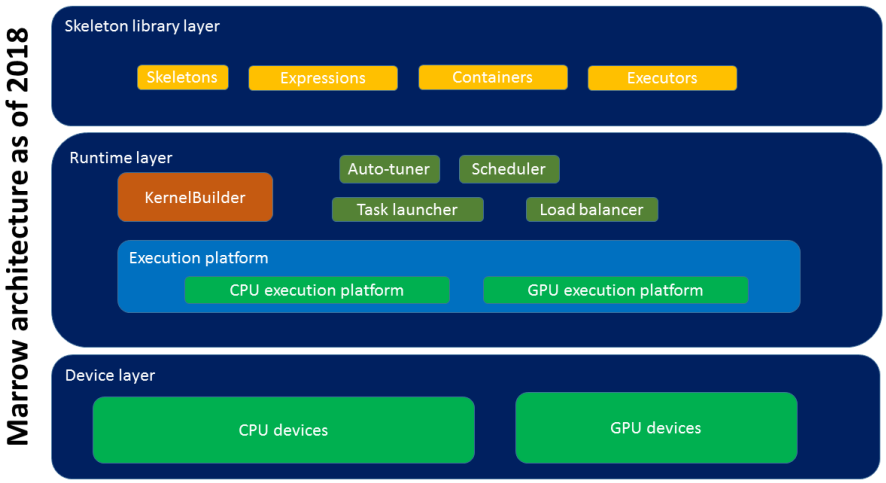

Figura 3.1: Diagrama mais detalhado da arquitetura do *Marrow*.

3.1.1 Camada biblioteca de esqueletos

3.1.1.1 *containers*

As estruturas de dados suportadas pelo *Marrow* onde são aplicadas as computações designam-se *containers*. O nome deriva da notação respetiva em C++[7] onde é referido que estas entidades assumem o papel de guardar coleções de outros objetos. Para além de guardar as referidas coleções, os *containers* devem gerir o espaço de armazenamento destas assim como aceder aos elementos. O acesso deve ser executado diretamente ou por iteradores. Como o *Marrow* encontra-se implementado em C++, sendo esta uma linguagem com orientação a objetos, existe um mecanismo de herança entre classes pelo que todos os *containers* implementados herdam de uma classe principal. Esta classe é designada por *base_container* onde estão contidas as operações em comum, sendo estas operações:

- *size*: Devolve o tamanho da estrutura ou o número de elementos presentes dependendo se a estrutura for estática ou dinâmica;

- *getters* e *setters*: Acesso e manipulação de um elemento guardado na estrutura cujo índice corresponda ao que deve ser dado;
- *fill*: Preenchimento de todos os elementos presentes com um dado valor;

Os *containers* herdeiros do *base_container* contêm ainda operadores cuja implementação é específica, sendo eles o operador de afetação e de *overload*. O operador de afetação permite obter a estrutura que está no lado esquerdo da expressão (*Left hand side*) a partir do conteúdo presente no lado direito desta (*Right hand side*). Se no *right hand side* estiver uma outra estrutura equivalente, o operador em questão permite a movimentação do conteúdo desta para a que está no *left hand side*. O operador de *overload* permite obter uma referência para o elemento na posição a indicar como argumento. Tendo em conta o que foi referido, até ao momento em que a tese foi elaborada o *Marrow* suportava os seguintes *containers*:

- *arrays*: Estruturas de dados estáticas unidimensionais ;
- matrizes: Estruturas bidimensionais que similarmente ao *arrays* são estáticos;
- vectores: Estas estruturas são unidimensionais no entanto ao contrário dos *arrays* são dinâmicos;

Um container é instanciado através de um conjunto de argumentos de *template*. Sobre este conjunto, é necessário passar o tipo dos elementos a guardar (inteiro, float,...) e o tamanho da coleção se for estático, podendo ser uma variável constante do tipo *size_t*[2]. Existem argumentos adicionais como a unidade de partição, a estratégia de alocação e a vizinhança de cada partição acedida por um *worker*[3] que têm valores por omissão. No entanto estes argumentos não revelaram ser tão importantes como os primeiros que foram referidos.

À exceção dos vetores, as estruturas na generalidade são estáticas no sentido em que para serem instanciadas, as dimensões da estrutura devem ser fornecidas em tempo de compilação como argumentos de *template*. De acordo com as normas em C++ não é permitido passar variáveis não constantes como argumento de *template*, o que no âmbito da tese corresponde a um obstáculo à resolução do problema. Para usar as estruturas estaticamente seria necessário conhecer em tempo de compilação, para um dado cenário de *docking*, variáveis que dependem das dimensões das proteínas envolventes, o que não é possível.

A implicação para a versão final seria esta funcionar apenas para proteínas com uma dada dimensão o que não é o pretendido. Neste contexto foi necessário implementar versões dinâmicas destas estruturas, sendo os detalhes esclarecidos mais adiante no documento (secção 4.4.4).

[2] Do standard C++11 para a frente existe o tipo *size_t*. Este tipo é equivalente a um inteiro, sendo adequado para indexação de coleções e em ciclos[52].

[3] *Workers* no contexto de computação paralela é o termo designado para as *threads* lançadas pela CPU ou GPU no momento de execução. Idealmente estas unidades lançam um *worker* por cada *core* à disposição.

3.1.1.2 Skeletons

atualmente o *Marrow*, através dos seus *skeletons*, suporta diretamente seis padrões de paralelismo: *Scan*,*Filter*,*Map*, *Reduce*, *Loop* e *Pipeline*. No âmbito da corrente tese apenas foi necessário recorrer ao *Reduce* e ao *Map*, pelo que estes serão os únicos a serem abordados no presente documento. Michael McCool (2010)[27] abordou as definições para ambos redução e mapeamento. Uma redução consiste na aplicação de uma operação associativa a todos os elementos de uma coleção, sendo aplicada em pares de elementos tal como ilustrado na figura 3.2.

O resultado será tal como o nome sugere uma redução de cada par a um elemento individual. Deste modo reduções aplicadas a elementos constituintes de uma estrutura de dados com D dimensões tem como resultado final uma estrutura com $D-1$ dimensões. Uma redução aplicada a um *array*, uma estrutura com apenas uma dimensão resulta num escalar, sendo esta situação o limite até onde pode ser aplicada a redução.

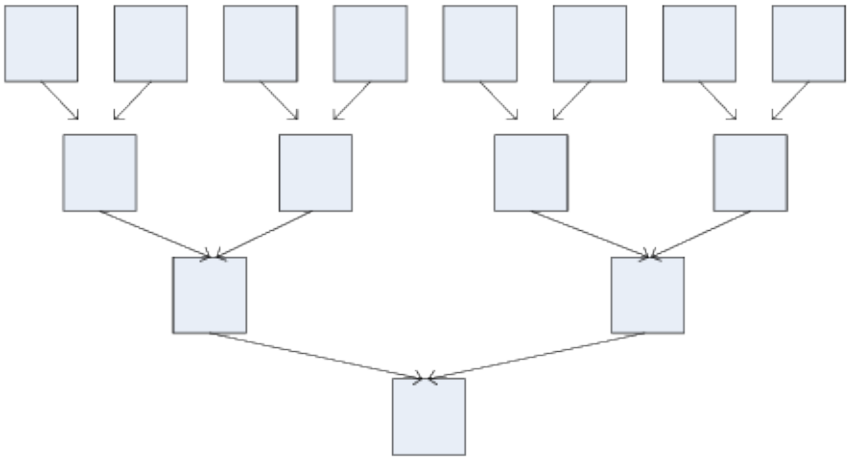

Figura 3.2: Esquema demonstrativo de uma redução[14].

Uma operação de *map* por sua vez define-se como a aplicação de uma função a definir pelo programador para um dado conjunto de elementos, criando uma instância do mesmo conjunto com as alterações efetuadas (figura 3.3). A ordem com que os elementos são alterados é irrelevante pois cada uma das alterações é aplicada de forma independente, pelo que o padrão revela ser adequado para execuções em paralelo de computações cujo problema pode ser dividido em tarefas.

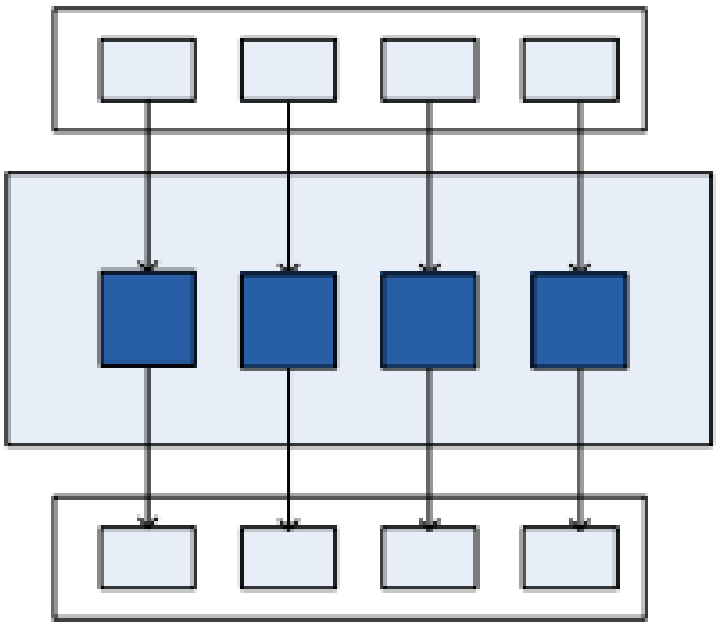

Figura 3.3: Esquema demonstrativo de um mapeamento em paralelo[14].

3.1.1.3 Expressões

As expressões do *Marrow* representam os nós e folhas da árvore de computação a passar para a camada de *runtime*. atualmente estão implementadas as expressões:

- Afetação: expressão que permite estabelecer um valor para uma dada variável. A sua declaração no *Marrow* é indicada através do operador ('=') de forma a garantir homogeneidade com a linguagem C++;
- Chamada: com esta expressão é efetuada uma chamada para uma dada função, através do operador ('()');
- Acesso de dados: Efetua o acesso aos conteúdo de um elemento de um dado *container*, tendo como requisito o índice onde o elemento se encontra;
- Escalar: Permite obter um valor escalar a partir de um *container* unidimensional, usando uma redução;

Para além dos operadores referidos, o *Marrow* suporta os operadores aritméticos e lógicos comuns que podem ser encontrados na maior parte das linguagens funcionais existentes. Em termos de aridade, os operadores podem ser:

- Unários: Operadores que afetam apenas uma estrutura de dados;
- Binários: Ao contrário dos operadores unários, os binários afetam duas estruturas de dados. Exemplos incluem os operadores aritméticos;
- Ternários: Operadores que necessitam de três operados para serem executados ;

3.1.1.4 Aninhamento de árvores de computação

Como foi referido no início do capítulo, uma das características do *Marrow* é a sua capacidade de aninhamento de árvores de computação. Esta funcionalidade é usada pelo programador quando recorre à palavra reservada do C++: auto [42] na sua solução. Um exemplo prático da implementação de uma função de distância euclidiana pode ser observada na listagem 3.1.

Listagem 3.1: Implementação de uma função de distância euclidiana no *Marrow*. De notar a composição de funções recorrendo à palavra reservada auto

```
1 float distance(float* coord1_ptr, float* coord2_ptr) {
2
3       array<float,3> coord1 (coord1_ptr);
4        array<float,3> coord2 (coord2_ptr);
5       auto subs = coord1 - coord2;
6       auto powers = subs * subs;
7       scalar<float> s = reduce<plus<float>>(powers);
8       return sqrt(s.value());
9 }
```

De notar que a ausência dos termos *auto* nas linhas 5 e 6 implicaria a geração de três árvores de computação, e por consequência, três *kernels* distintos. A presença destes termos no entanto permite ao *Marrow* gerar a árvore de computação presente na figura 3.4 e um único *kernel* será gerado. A linha 5 do código da função permite gerar uma árvore *subs*, contendo uma subtração entre as coordenadas de dois pontos. A linha 6 por sua vez gera uma segunda árvore *powers*, contendo uma multiplicação entre duas árvores *subs*. A raiz da árvore é criada na linha 7, onde é aplicada uma redução de soma ao resultado da árvore *powers*. Antes de gerar o código e retornar a raiz quadrada do resultado da computação (linha 8), o *Marrow* verifica primeiro se o código já tinha sido gerado. Em caso negativo o *Marrow* procede para a sua geração e execução através da árvore, senão executa diretamente.

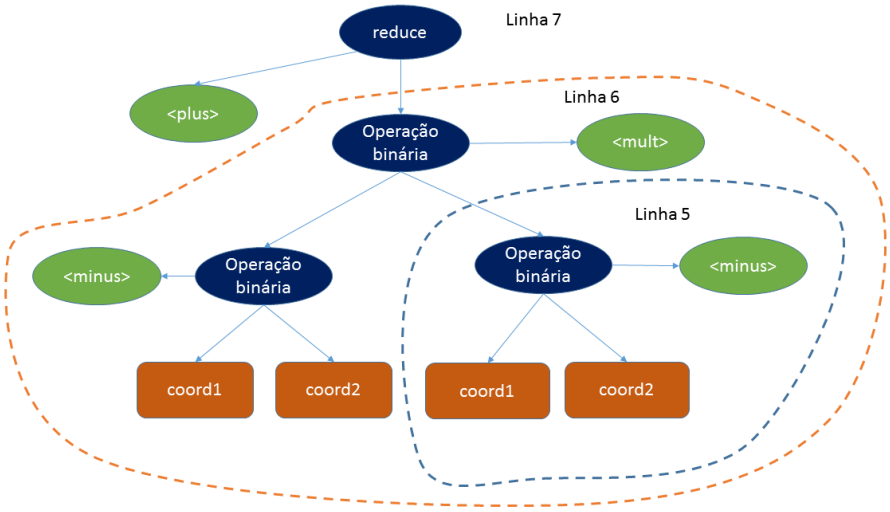

Figura 3.4: Para o código da listagem 3.1, o arrow gera esta AST.

3.2 Sumário

A *framework Marrow* foi discutida, dando relevância aos elementos que um programador pode utilizar para programar soluções para os seus problemas embaraçosamente paralelos e não a características como o modelo de execução por exemplo. No próximo capitulo serão discutidas as questões relacionadas com a deteção de problemas de desempenho e implementação das respetivas otimizações ao BiGGER.

Capítulo 4

BiGGER versão GPU

O presente capítulo tem como objetivo descrever detalhes relacionados a implementação de soluções para os problemas de desempenho do BiGGER, identificando regiões de código candidatas e a sua adequação para paralelizar na GPU. Sobre as regiões adequadas é abordada a implementação das respetivas otimizações recorrendo à biblioteca *Marrow*.

4.1 Detalhes iniciais

4.1.1 Ciclo de desenvolvimento

Os esforços para a implementação da solução para paralelizar o BiGGER seguiram um ciclo de desenvolvimento. Este ciclo denomina-se APOD (*Assess*, *Parallelize*, *Optimize*, *Deploy*) [32] e consiste em quatro fases (figura 4.1):

1. **Avaliar**(*Assess*) : Onde é feito uma avaliação (*assessment*) ao estado atual do programa, em termos de desempenho. Nesta fase são determinados os pontos do programa onde este passa mais tempo a executar e identificar os bottlenecks de instruções, através de profilers para confirmar as identificações efetuadas. No caso da dissertação, é feito o *profiling* do ficheiro bigger.lpi presente na pasta *bigger* da biblioteca *open-source* para o software que usa o BiGGER, Open-Chemera e determinadas as funções que este ficheiro chama onde a fração de tempo de execução é maior (*hotspots*) assim como zonas de código que atrasam a execução do programa (*bottlenecks*).

2. **Paralelizar**(*Parallelize*) : Após o *assessment* referido anteriormente estar concluído, procedeu-se para a fase de implementação do código para paralelizar os pontos encontrados na fase anterior. De acordo com *Six ways to SAXPY*, de Mark Harris [13], existem três possibilidades para implementar acelerações: usar bibliotecas com o

código que implementa as acelerações, diretivas OpenACC ou recorrer a linguagens para programação em GPUs, como CUDA ou OpenCL.

3. **otimizar**(*Optimize*) : Nesta terceira fase pretende-se adicionar melhorias à solução base para maximizar a performance, considerando as técnicas referidas na subsecção 2.2.3 e com inspiração nas abstrações relatadas na subsecção 2.3.

4. **Implantar**(*Deploy*) : A última fase do ciclo consiste em confrontar o desempenho obtido com as expectativas fundamentadas no início do ciclo. Se os resultados obtidos não corresponderem ao *speedup* potencial registado na fase inicial, é necessário voltar à fase *Assess*, recomeçando o ciclo.

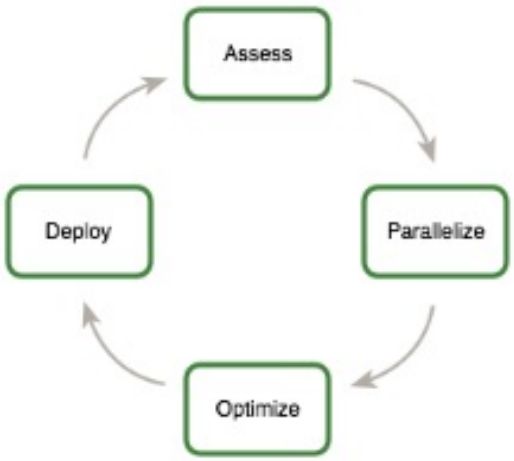

Figura 4.1: Diagrama do ciclo de desenvolvimento APOD[32].

4.1.2 Arquitetura geral

O desenvolvimento da solução ao desafio proposto na tese seguiu a arquitetura indicada na figura 4.2. A biblioteca Open-Chemera encontra-se dividida em diversas pastas entre as quais a *docking*, dentro da pasta *oclibrary* e a *sharedunits* que são as mais interessantes do ponto de vista da tese. Esta biblioteca contém componentes como o *dockprep* que é usado para construir toda a informação necessária para o BiGGER realizar o *docking*, sendo uma ferramenta importante para executar o *profiling* do algoritmo. O algoritmo pode ser executado a partir da interface gráfica ou da componente bigger presente na pasta com o nome respetivo, sendo que foi usada esta última alternativa no decorrer do trabalho. A arquitetura engloba ainda a biblioteca dinâmica *Marrow_protein_docking* resultante da compilação do *Marrow*. Nesta biblioteca encontra-se presente o código que resolve os problemas de desempenho detetados no Open-Chemera. As funções usam estruturas de dados e operações que estão incluídas na *framework Marrow*, que é integrada na biblioteca.

As duas componentes da arquitetura encontram-se implementadas em linguagens diferentes, sendo necessário a implementação de um elo de comunicação que permita

a adaptação das uma linguagem intermédia que o Pascal consiga entender. A segunda função deste elo consiste em estabelecer a ligação entre o Open-Chemera e a biblioteca. Mais adiante no documento será esclarecida a implementação deste elo, que na figura está representado nas entidades *Pascal wrappers* e *C wrapper*.

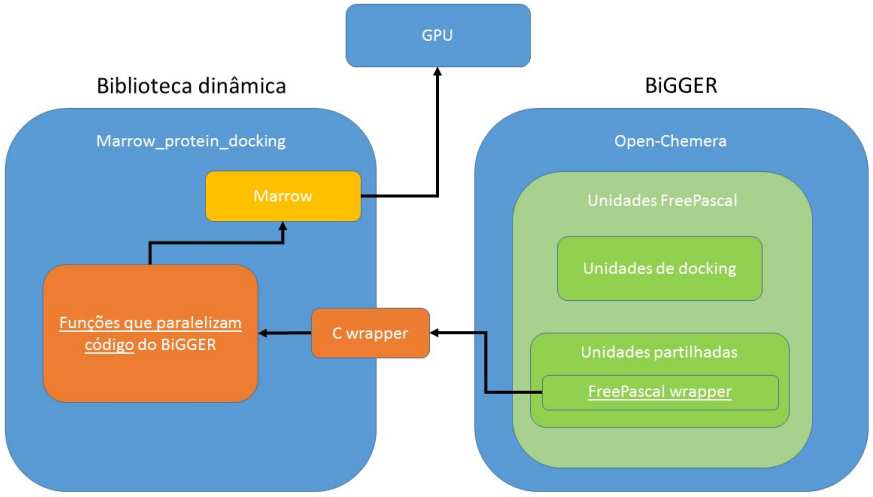

Figura 4.2: Diagrama ilustrativo para a arquitetura geral da solução.

4.2 Profiling

Nesta fase inicial do desenvolvimento das otimizações foi feita uma análise de performance ao BiGGER. A biblioteca Open-Chemera na versão sequencial foi submetida a um conjunto extenso de testes, variando parâmetros que o utilizador consegue passar ao BiGGER como o nº de átomos do ligando/recetor, a sobreposição mínima que as superfícies moleculares devem ter e o número de rotações no *docking* como está indicado na tabela 4.1. A ferramenta usada para a atividade foi a única que funcionou de diversas alternativas que podem ser usadas para fazer *profiling* a um programa que se encontra em Pascal, sendo a alternativa o programa para profiing Valgrind/Callgrind [43]. Este programa devolve um conjunto de dados que inclui o número total de chamadas efetuadas para uma dada função e duas métricas de avaliação de custo temporal: *self cost* e *inclusive cost*. Para efeitos de amostragem de resultados, foi considerada principalmente a métrica *self cost* pois esta refere-se à fração de tempo de execução que é passado dentro da função, enquanto que a *inclusive cost* inclui as frações de tempo das funções chamadas pela função a considerar. No caso deste programa, após a execução do *profile* é gerado um ficheiro específico com os resultados que podem ser consultados através de um segundo programa de nome KCacheGrind.

Sólido geométrico	**Átomos Probe**	**Átomos Target**	**MinOverlap** (Sobreposição mínima)	**Rotações**
Cubo	216	512	0	100
Esfera	1728	3375	500	1000
-	6859	19683	1000	1500
-	19683	32678	2000	2000
-	-	-	200000	6000
-	-	-	-	15000

Tabela 4.1: Parâmetros de teste e variação.

4.2.1 Sobre os parâmetros de teste

4.2.1.1 Complexos de proteínas usados

Para que seja possível determinar o impacto da forma geométrica dos pares na performance do BiGGER, foram considerados pares ligando-recetor que assumam formas uniformes, mais precisamente cubos e esferas. Adicionalmente, não foram testadas situações em que, por exemplo, o ligando tem forma cúbica e o recetor esférica, pois isto implica duplicar a quantidade de testes necessários.

Em termos de número de átomos, não foram aplicados testes em que o número de átomos do recetor seja menor do que o número de átomos do ligando, tendo em conta que não correspondem a uma situação real assim como introduzem esforços adicionais ao BiGGER que são desnecessários. Conforme foi referido no capítulo 2 o BiGGER resolve o problema de *docking* através de um conjunto de passos, entre eles a digitalização da superfície molecular para grelhas tridimensionais. Para cada *job* solicitado, é criada e digitalizada uma vez a grelha do recetor e tantas vezes como rotações pedidas a grelha do ligando.

4.2.1.2 Sobreposição mínima e rotações

Com o parâmetro sobreposição mínima, o utilizador define o limite mínimo para a sobreposição entre as superfícies moleculares do ligando e do recetor que uma dada pose deve ter para não ser descartada na fase de filtragem de possibilidades do BoGIE. Adicionalmente o utilizador consegue definir o número de rotações que o BoGIE deve iterar para completar um *docking*, sendo cada rotação correspondente a uma pose. Foi pretendido estudar o impacto da variação da sobreposição mínima na fase de *scoring* do algoritmo, sendo que esta foi incrementada de forma gradual entre o valor nulo e um valor excessivamente grande para uma situação real. Sobre as rotações, foi aplicado o mesmo procedimento, sendo que neste caso o número de rotações foi variado gradualmente entre 100 rotações (valor baixo para uma situação real) e 6000/15000 (valores usados habitualmente num *docking*).

4.2.2 Scripts auxiliares desenvolvidos

Tendo em conta as variações de parâmetros referidas na tabela 4.1, o leitor fica sensibilizado para a extensão de cenários de teste possíveis de realizar, produto do número de combinações possíveis. Para cada um dos pares de número de átomos (216 do ligando e 512 do recetor por exemplo), foram feitos 5 testes a variar o *minOverlap* e dentro de cada uma destas variações 6 testes a variar as rotações. No total foram realizados 120 testes, em que para cada um foram necessários aplicar passos manuais como:

1. A criação, através da sub-componente dockprep do BiGGER, do ficheiro de *job* (xml) para o *docking* sendo este último completado pela execução do BiGGER;

2. Alteração dos campos necessários do ficheiro xml para o *docking* estar dentro do cenário pretendido;

3. Executar o teste por linha de comandos;

4. Guardar os resultados numa folha Excel.

Todo este processo para cada um dos 120 testes fez com que a tarefa de realizar o *profile* ao BiGGER tenha demorado excessivamente. Como tal foram implementados 3 scripts permitem automatizar o procedimento de testar quer o programa inicial, quer as otimizações efetuadas. Adicionalmente têm o efeito de diminuir o erro humano na execução dos passos referidos. O primeiro script (*DummyJobCreator*) consiste num criador de ficheiros xml de *jobs* para o BiGGER executar, replicando a estrutura dos ficheiros equivalentes mas gerados pelo dockprep. Os procedimentos encontram-se esclarecidos no anexo B.

O segundo script é meramente um ficheiro com uma sequência de comandos para um terminal Linux. As instruções seguem a sequência de invocar para cada um dos 120 testes o *dummyJobCreator.py* com os respetivos argumentos, seguido de instruções para executar o teste usando o *profiler*. Os passos finais do script são instruções para criar pastas onde estarão os resultados de cada um dos testes e adicionalmente as instruções para mover ambos ficheiro de *job* e de resultado de *profiling* para dentro da pasta respetiva.

O terceiro programa(*LinearGraphs*) que similarmente ao primeiro foi desenvolvido em Python, cria gráficos lineares com os valores de *self cost* obtidos pela execução do bigger com o valgrind/callgrind para o conjunto de funções mais problemáticas no OpenChemera. O programa faz a leitura de uma matriz com estes valores, gerando três agrupamentos de gráficos: (i) variação por número de átomos do ligando e recetor, (ii) variação por sobreposição mínima, (iii) variação por rotações. Um exemplo de output do script pode ser observado na figura 4.3.

Estes scripts automatizam a maioria do *workflow* necessário para aplicar o *profiling*. Falta no entanto automatizar a introdução/ atualização dos valores de *self cost* na matriz que o segundo script usa para gerar os gráficos, tarefa esta que até ao momento revelou-se difícil de concretizar.

4.2.3 Conclusões

Verificou-se que os problemas de desempenho estão localizados em duas unidades partilhadas[4] chamadas *geomhash* e *geomutils* do Open-Chemera. Na figura 4.4 está ilustrado o troço de execução para o qual o BiGGER demora mais tempo a executar e por consequência aquele onde foram aplicadas as paralelizações prioritárias, sendo este troço referente à fase do BiGGER onde são criadas as grelhas tridimensionais dos pares envolventes no *docking*. Existem duas funções: *setExtremes* e *addModel* em que o *self cost* retornado teve um valor considerável. No entanto os valores de *self cost* demonstraram que estas funções são problemáticas apenas em situações específicas. De seguida apresentam-se, com base nos gráficos presentes em A.1, as conclusões referentes ao impacto que cada um dos argumentos manipulados durante o *profiling* ao algoritmo tem sobre este.

4.2.3.1 Impacto do número de rotações/tamanho das estruturas

Comparando as proporções de tempo de execução para cada uma das etapas de *docking*, a maioria ou até mesmo a totalidade do tempo de execução é gasto na digitalização para grelhas tridimensionais. Os resultados indicam que as funções com as percentagens de *self cost* mais elevadas têm os nomes *isInnerPoint* e *distance*, presentes na *geomhash* e *geomutils* respetivamente.

De notar que as percentagens de *self cost* para as funções referidas assim como o tempo de digitalização (e por sua vez o de execução) são crescentes com o aumento do número de rotações e do tamanho das estruturas. Tendo em conta esta observação, considerou-se como objetivo principal a paralelização das funções *isInnerPoint* e *distance* de forma a oferecer ao algoritmo escalabilidade com o número de rotações/tamanho das proteínas e reduzindo a proporção de tempo de execução gasto na digitalização para grelhas tridimensionais.

4.2.3.2 Impacto da sobreposição mínima

Pelos gráficos em que é variado o *minOverlap*, é possível observar que a curva mantém-se constante para a maior parte das situações, variando apenas a curva respetiva à função *addModel*, em que para um valor de *minOverlap* nulo o *self cost* para esta função demonstrou ser bastante superior ao mesmo para situações em que o *minOverlap* é maior que nulo. Deste modo concluiu-se que a curva assume-se como decrescente em relação ao *self cost*.

Este comportamento da curva é justificável pois para valores de *minOverlap* baixo, o BiGGER pode passar uma parcela de tempo considerável a preencher uma lista de modelos solução com soluções de poses em que o valor de *score* associado à sobreposição é maior ou igual a zero até preencher por completo a lista. Se a sobreposição mínima for aumentada, a probabilidade do algoritmo encontrar soluções adequadas para reservar

[4] As unidades designam-se partilhadas pois são usadas por mais do que uma unidade de *docking*.

na lista baixa, sendo que a lista já não é preenchida e o *self cost* da função *addModel* baixa. A mesma probabilidade é zero em situações onde a sobreposição mínima definiu-se como o valor máximo estabelecido para a bateria de testes. Como tal infere-se que o parâmetro sobreposição mínima tem peso apenas para a fase de *scoring* do algoritmo e não para a digitalização para grelhas tridimensionais, pelo que foi decidido não introduzir paralelizações para a função *addModel*.

4.2.3.3 Impacto do formato das estruturas

Existe ainda a função *setExtremes*, cujos valores de *self cost* registados não são tão consideráveis como os valores das funções referidas nos pontos anteriores. Sobre os seus valores de *self cost*, acrescenta-se que as curvas para a *setExtremes* nos gráficos são constantes em todos os gráficos nos 3 agrupamentos, pelo que o comportamento desta função não é afetada por nenhum dos parâmetros. No entanto foi verificada a diferença dos valores de *self cost* nos testes em que foram usados complexos com formato cúbico para os com formato esférico. Deste modo conclui-se que apenas o formato dos complexos afeta esta função, desta forma optou-se por não paralelizar a *setExtremes*.

4.2.3.4 Tendência das percentagens de *self cost*

Este último ponto diz respeito aos limites máximos de valores de *self cost* observados para as funções referidas. Para os cenários de teste mais exigentes verificou-se que o *self cost* do BiGGER encontrava-se a 96% e os restantes 4% dizem respeito às bibliotecas externas auxiliares, sendo que esta fração de tempo não pode exceder ou alcançar os 100%. Com base neste facto e na análise dos gráficos gerados, é possível estimar a assimptota horizontal das curvas associadas às funções de distância e *isInnerPoint*, que está situada entre os 40% e 45% para a primeira e 35% e 40% para a última.

Esta inferência poderia ser comprovada introduzindo cenários de teste que excedem a exigência máxima considerada para esta fase (aumentar o tamanho dos pares e mais rotações com sobreposição mínima nula), certamente haveria um resultado em que a fração do BiGGER atingisse os 99%. No entanto para os cenários atuais mais exigentes, o tempo de execução de cada um demorou aproximadamente 8 horas a executar, o que faz com que a execução de um teste ainda mais exigente possa demorar até 24H a terminar, não sendo compensável face à alternativa de estimar os limites.

4.2.3.5 Construção grelha base vs grelhas superfície-centro

Além das conclusões anteriores, os resultados permitiram inferir que a fase de construção da grelha base tem um desempenho pior do que a determinação das regiões centrais e de superfície da proteína. Esta afirmação é justificável pois a determinação da superfície/região central é feita deslocando uma cópia da grelha base nas 26 direções adjacentes e aplicando uma operação XOR entre os nós de ambas as grelhas. Apesar de serem feitas 26 deslocações, o processamento é menos exigente comparando com o que é feito na

construção da grelha base, onde para cada ponto da grelha é calculada da distância a todos os átomos da proteína. A distância é depois comparada com o valor da distância de VdW para o átomo corrente, na eventualidade de ocorrer apenas uma situação em que seja menor e o nó corrente é classificado como interno, caso contrário é externo.

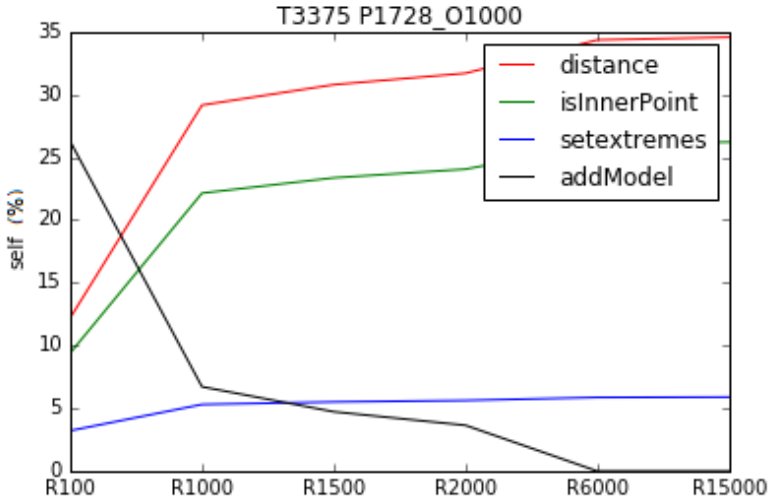

Figura 4.3: Exemplo do output do script *LinearGraphs*

4.2.4 Descrição do fluxo de execução problemático

Por observação do código-fonte presente no Open-Chemera infere-se o pseudocódigo presente nas listagens 4.1 e 4.2 assim como a figura 4.4. Todos os elementos referem o fluxo de execução do Open-Chemera onde o *profiler* determinou ter pior desempenho. Na versão sequencial do Open-Chemera, quando o BiGGER executa, é iniciado um procedimento[3] *RunJobs* (linhas 2-17 do pseudocódigo 4.1), em que para cada *job* é lido um conjunto de parâmetros que caracterizam o cenário de *docking*. Para além dos parâmetros na secção anterior sobre o *profiling*, o programa lê e guarda as coordenadas dos átomos de ambos ligando e recetor.

O fluxo principal do *docking* é executado num procedimento de nome *RunGeometric* (linhas 4-15). Neste procedimento é feita a digitalização da superfície molecular do recetor, sendo esta feita uma vez para cada *job* solicitado (linha 8). No *RunGeometric* também são iteradas todas as rotações que o ligando possa ter em relação ao recetor, de forma a completar o *job* corrente. De notar que um *job* é considerado completo quando o número de rotações iteradas com o *docking* completo e o *score* respetivo calculado é igual ao número de rotações solicitada. Para assegurar este comportamento, o procedimento contém um loop com uma condição de guarda onde é chamada uma função de nome *Dock*(linha 19 da listagem 4.1).

[3]Um procedimento é o equivalente em Pascal a uma função/método do tipo void.

O algoritmo BoGIE relatado no capítulo 2 é iniciado com a invocação da função *Dock*. No mesmo capítulo é referido que este algoritmo é responsável pela execução da primeira fase de um *docking* executado pelo BiGGER, ou seja, a determinação da superfície molecular do par. Esta etapa encontra-se implementada na função *Dock*. Para fazer a digitalização de ambos recetor e ligando, os procedimentos *BuildTargetGrid* e *BuildProbeGrid* são chamados pelo *RunGeometric* e pela função *Dock*, respetivamente.

A digitalização das proteínas para grelhas tridimensionais envolve a construção inicial de uma grelha base onde a partir das distâncias de VdW de cada átomo do ligando/recetor é determinado se cada ponto da grelha corresponde à região interna da proteína ou à externa. A partir da grelha base, são construídas a grelhas de pontos centrais e de superfície. Ambas as construções estão implementadas num procedimento denominado *BuildFromSpheres*. Este procedimento invoca dois procedimentos para determinar o reconhecimento da superfície molecular da proteína através da digitalização para grelhas tridimensionais: *BuildBaseGrid* responsável por determinar a grelha base e *BuildSurfCoreGrids* para determinar as grelhas central e de superfície.

Adicionalmente o algoritmo avalia a sobreposição no contacto entre as duas entidades. Se for superior ou igual ao valor solicitado para a sobreposição mínima, a pose é guardada numa lista com as poses mais adequadas. Esta funcionalidade está implementada na função *addModel*.

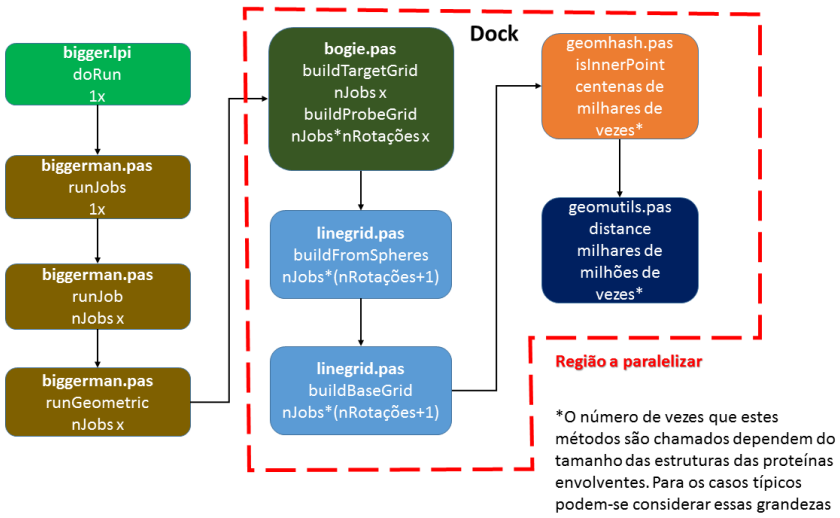

Figura 4.4: Grafo de chamadas para o troço de execução do BiGGER onde estão localizados os *hotspots* que foram identificados por análise aos resultados do *profiling*. Nos blocos estão presentes as unidades/ficheiros assim como o número de vezes que os métodos são chamados e quem chama.

Listagem 4.1: Pseudocódigo para o troço que inicializa a digitalização da superfície molecular para grelhas tridimensionais

```
void runJobs(){
  para cada job pedido{
    // runJob
    indexJob++;
    if(rotacaoCorrente < #rotacoes)  {
    runGeometric(indexJob);
    }
    //Fase de scoring
    ...
  }
}
void runGeometric(int index){
  //carregar estruturas e obter átomos do ficheiro de jobs
  //para ambos target e probe assim como as distâncias de VdW destes;
  buildTargetGrid(coordAtomosTarget, distVdWTarget);
  construir rotacoes;
  while Dock(coordAtomosProbe,distVdWProbe){

  }
}
boolean Dock(float*[3] pontosProbe,float* distanciasPontosProbe){

  while(rotacaoCorrente < ultimoIndiceRotacaoCorrente){
    rotacaoCorrente++;
    buildProbeGrid(pontosProbe, distanciasPontosProbe, rotacaoCorrente);
    // Avaliacao do contacto entre os átomos do ligando e do recetor
    ...
    addModel(scoreSobreposicao)
  }
  return rotacaoCorrente==ultimoIndiceRotacaoCorrente;
}

void addModel(int scoreSobreposicao, int X, int Y, int Z){
    if(scoreSobreposicao > sobreposicaoMínima){
      if(lista de possibilidades cheia
      && scoreSobreposicao > scoreSobreposicao pior possibilidade){
        //Descartar pior candidato
      }
      //AdicionarPossibilidade
    }
}
```

Listagem 4.2: Pseudocódigo para a digitalização para grelhas tridimensionais

```
void buildProbeGrid(float*[3] pontosProbe
,float* distanciasPontosProbe, Quaternion rotacao){
    pontosTmp = rotate(rotacao, pontosProbe);
    buildFromSpheres(pontosTmp, distanciasPontosProbe);

}
void buildTargetGrid(float*[3] pontosTarget, float* distanciasPontosTarget) {
   buildFromSpheres(pontosTarget, distanciasPontosTarget);
}
void buildFromSpheres(float*[3] coordAtomos, float* distanciasVdW){
  buildBaseGrid(coordAtomos);
    //Construcao da grelha central-superfície
}
void buildBaseGrid(float*[3] coordAtomos, float* distanciasVdW){
  int TLineSegment [2];
  int TGridLine [][2];
  int TGridPlane[][][2];
  resolucao = 1.0;
  metade = resolucao / 2;
  para um x a variar entre 0 e o limite x da grelha{
    xPontoCorrente = x*Resolucao + metade;
    para um y a variar entre 0 e o limite y da grelha{
      yPontoCorrente = y*Resolucao + metade;
      para um z a variar entre 0 e o limite z da grelha{
        zPontoCorrente  = z*Resolucao + metade;
        // Se a distância euclidiana entre o ponto e todos os pontos
        //  da nuvem fPoints for < distancia fRads para cada ponto desta
        if( isInnerPoint(PontoCorrente, fPoints[indice]))
          ZLine[z] = 1;
          c.c
          ZLine[z] = 0;
      }
     TGridPlane[x,y] = IntegersToLine(ZLine);
    }
  }
}
```

4.3 Análise de candidatos à paralelização

Tendo em conta as conclusões apontadas, em quatro funções identificadas no código do Open-Chemera existem duas que se apresentam como boas candidatas à implementação de paralelizações no GPU: *distance* e *isInnerPoint*. Considerou-se paralelizar estas funções crucial para acelerar a digitalização das proteínas para grelhas tridimensionais e por consequência o algoritmo. Para além destas funções, existe o procedimento *BuildBaseGrid*, que usa ambas as funções referidas anteriormente.

4.3.1 Função distance

Esta função recebe dois conjuntos de coordenadas tridimensionais de dois pontos e determina a distância euclidiana entre estes (listagem 4.3). A função é paralelizável pois a expressão da distância euclidiana é divisível em cada um dos termos.

Listagem 4.3: Pseudocódigo para a função de distância euclidiana

```
1 float distance(float[3] Vec1, float[3] Vec2);
2 {
3   result=Sqrt(Sqr(Vec1[0]-Vec2[0])+Sqr(Vec1[1]-Vec2[1])+Sqr(Vec1[2]-Vec2[2]));
4   return result;
5 }
```

4.3.2 Função *isInnerPoint*

A função *isInnerPoint* tem o objetivo de determinar a distância de um ponto da grelha a cada um dos átomos da proteína para a qual a digitalização para grelhas tridimensionais está a ocorrer, comparando de seguida a distância obtida com a distância de VdW (*FRads*) para o átomo corrente. A função determina o ponto como interno se a distância obtida for menor do que o valor presente no FRads e externo em caso contrário.

Por análise ao código da função (próximo do que está na listagem 4.4) é possível reparar que existem quatro ciclos aninhados (três ciclos para as coordenadas xyz e um para uma variável f). Estes ciclos são usados para iterar um conjunto de índices que apontam para as coordenadas atómicas correntes, estando estes índices guardados numa coleção (fHashGrid). Esta estrutura tem quatro dimensões e guarda a dispersão dos átomos em cada uma das fatias da grelha. Foi tomado como conclusão que a complexidade desta função cresce com o número coordenadas atómicas (fPoints) para avaliar cada célula da grelha. A função é paralelizável pois o facto de a computação ser aplicada para todos os átomos, sendo cada um destes tratado de forma iterativa faz com que a função seja embaraçosamente paralela.

A versão sequencial está otimizada para terminar a sua execução no momento em que determina pela primeira vez que o ponto corrente da grelha está dentro do raio de VdW de um átomo, efetuando menos computação no CPU. Outro aspeto de desenho relevante é o facto de para cada segmento, o BoGIE não considerar todos os átomos da grelha mas os que estão localizados apenas na fatia da grelha respetiva. Neste contexto os segmentos que estejam localizados em fatias onde não existam átomos são imediatamente considerados externos. Adaptar ambos os comportamentos para a versão GPU não é uma tarefa simples, pelo que a versão paralelizada irá efetuar mais computações do que a versão CPU.

4.3.3 Procedimento BuildBaseGrid

Na listagem 4.5 está presente a versão sequencial do pseudocódigo para este procedimento. De notar que no âmbito de construção da grelha base, a cada célula da grelha é aplicada uma conformação às suas coordenadas em função de um valor de resolução (tipicamente 1.0f). Depois é verificado se a célula corresponde a uma região interna ou externa, usando ambas funções *isInnerPoint* e *distance*. No total existem 3 ciclos aninhados que permitem iterar todas as células da grelha, no loop interno onde é variada a coordenada z são executados ciclos adicionais por parte da função *isInnerPoint* para avaliar a

Listagem 4.4: Pseudocódigo para a função *isInnerPoint*

```
void GridBounds(out B1, B2: Integer; const Val: TFloat;
  const Hi: Integer){
  B1 =Trunc(Val)-1;
  B2:=Trunc(Val)+1;
  if B1>Hi then B1:=Hi;
  if B1<0 then B1:=0;
  if B2>Hi then B2:=Hi;
  if B2<0 then B2:=0;
}
float[3] Simmetric(Vec: TCoord){
 Result = -Vec;
}
boolean isInnerPoint(float * pontoCorrente, float fPoints*[3]){
  float[3] tmpc,FShiftToGrid, maxc, minc;
  float GridStep = Max(Max(FRads),1.0f);
  FShiftToGrid = Simmetric(minc) ;
  FHighX:=Trunc((maxc[0]-minc[0])/GridStep);
    FHighY:=Trunc((maxc[1]-minc[1])/GridStep);
  FHighZ:=Trunc((maxc[2]-minc[2])/GridStep);
  tmpc = Multiply(Add(C,FShiftToGrid),(1/GridStep));
  GridBounds(x1,x2,tmpc[0],FHighX);
    GridBounds(y1,y2,tmpc[1],FHighY);
    GridBounds(z1,z2,tmpc[2],FHighZ);
  resultado = falso;
  para um x a variar entre x1 e x2 {
    para um y a variar entre y1 e y2{
      para um z a variar entre z1 e z2{
        para um f a variar entre 0 e
        o índice mais alto de fHashGrid[x,y,z]{
            indice = fHashGrid[x,y,z,f];
             if distance(pontoCorrente,fPoints[indice])
             < fRads [indice]
              resultado = verdadeiro;
              break;
             cc
              resultado = falso;
        }
      se resultado == verdadeiro break;
      }
    se resultado == verdadeiro break;
    }
  se resultado == verdadeiro break;
  }
}

float distance(float[3] Vec1, float[3] Vec2);
{
  result=Sqrt(Sqr(Vec1[0]-Vec2[0])+Sqr(Vec1[1]-Vec2[1])+Sqr(Vec1[2]-Vec2[2]));
  return result;
}
```

Listagem 4.5: Pseudocódigo para o BuildBaseGrid

```
void buildBaseGrid(float*[3] coordAtomos, float* distanciasVdW){
  int TLineSegment [2];
  int TGridLine [][2];
  int TGridPlane[][][2];
  resolucao = 1.0;
  metade = resolucao / 2;
  para um x a variar entre 0 e o limite x da grelha{
    xPontoCorrente = x*Resolucao + metade;
    para um y a variar entre 0 e o limite y da grelha{
      yPontoCorrente = y*Resolucao + metade;
      para um z a variar entre 0 e o limite z da grelha{
        zPontoCorrente  = z*Resolucao + metade;
        // Se a distância euclidiana entre o ponto e todos os pontos
        //  da nuvem fPoints for < distancia fRads para cada ponto desta
        if( isInnerPoint(PontoCorrente, fPoints[indice]))
          ZLine[z] = 1;
          c.c
          ZLine[z] = 0;
      }
     TGridPlane[x,y] = IntegersToLine(ZLine);
    }
  }
}
```

célula com cada uma das coordenadas dos átomos. Todos estes pontos fazem com que o procedimento seja paralelizável, sendo um bom candidato para paralelização em GPU.

4.4 BiGGER paralelizado

4.4.1 Wrappers C e Pascal

O ponto de partida para a implementação da abordagem final consistiu em definir como invocar e executar código C++ da biblioteca dinâmica gerada pelo *Marrow* (*Marrow_protein_docking*) nas funções e procedimentos do Open-Chemera onde é pretendido aplicar paralelizações. Uma possibilidade encontra-se descrita no artigo ''How to use C code in Free Pascal projects" [23] de Gilles Marcou, Etienne Engler e Alexandre Varnek. Este artigo descreve uma forma de chamar código não só C como também C++. Sobre a parte da solução implementada em C++, o artigo aponta para a necessidade de usar uma entidade que sirva de interface entre o código C++ e Pascal. O motivo consiste no facto do ambiente Pascal não conhecer objetos C++ nem primitivas como por exemplo *arrays* ou strings.

De forma análoga em C/C++ não são reconhecidas as estruturas dinâmicas (*arrays* dinâmicos[4]) que fazem parte do Pascal, pelo que é necessária garantir a adaptação para apontadores de C/C++[5]. Deste modo o papel da entidade consiste em adaptar o código

[4]O Pascal suporta *arrays* dinâmicos, sendo o *array* habitual com a diferença de as dimensões são desconhecidas em tempo de compilação[8].

[5]Os apontadores do C/C++ são equivalentes aos *arrays* dinâmicos do Pascal

C++ para código C, sendo este último reconhecido pelo Pascal. Sobre a adaptação referida, são mencionadas no artigo duas alternativas em relação à maneira como implementamos esta entidade denominada *C wrapper*, sendo que foi optada a segunda alternativa por ser apontada pelos autores do artigo como a mais simples.

Com o intuito de averiguar se a metodologia defendida pelo artigo encontra-se correta, os procedimentos de implementação que o documento aborda foram seguidos. Desta averiguação resultaram dois artefactos de código: uma versão própria do *hello world* referida no artigo e um programa que calcula expressões aritméticas.

Foi implementada no *Marrow_protein_docking* para além do código a partir do qual serão gerados os *kernels*, uma entidade *C wrapper* onde constam todas as funções implementadas que geram os *kernels* necessários para otimizar o desempenho do BiGGER. Adicionalmente é necessário aplicar alterações no código Pascal. À semelhança do que foi aplicado para o código da biblioteca, o artigo refere a necessidade de um *wrapper* em Pascal que define as funções em C/C++ a chamar no projeto, sendo estas definições acompanhadas com uma palavra reservada *cdecl*[1]. Também é necessário um conjunto de instruções pré-processador para estabelecer a ligação com a biblioteca.

Sobre os artefactos referidos anteriormente, de notar que ambos têm uma característica em comum: existe uma unidade Pascal que serve de programa principal e invoca as funções presentes numa outra unidade *Pascal wrapper*. Abstraindo esta característica para a solução, decidiu-se não criar uma unidade Pascal *wrapper* mas sim incluir as definições e as instruções pré-processador na unidade partilhada destinada a questões de digitalização para grelhas tridimensionais: *geomutils*. Deste modo esta unidade assume o papel de Pascal *wrapper* na arquitetura da solução. A justificação para esta escolha consiste no facto de as funções e procedimentos da *geomutils* serem usadas por múltiplas unidades que implementam a etapa de digitalização para grelhas tridimensionais, em particular a unidade *linegrids* que invoca a função *isInnerPoint*. Esta abordagem garante facilidade de deteção de erros de compilação relacionados com a ligação à *Marrow_protein_docking* assim como de manutenção de código.

4.4.2 Função distance

Tendo em conta que uma das funções onde o *profiling* detetou as percentagens mais elevadas de *self cost* foi a função de distância euclidiana, a primeira otimização implementada para o BiGGER consistiu em implementar no *Marrow* uma versão paralelizada desta função de forma a reduzir as percentagens referidas. De notar que a função é auxiliar, sendo usada na função *isInnerPoint* para calcular a distância entre de ponto da grelha tridimensional a um átomo da proteína, através das suas coordenadas.

[1] A palavra reservada *cdecl* permite-nos como programadores passar a ambos compilador e ligador a informação de dever processar os argumentos da função como se estivessem em ambiente C. Não incluir esta palavra reservada na frase pode levar ao termino abrupto da execução do programa.

Listagem 4.6: Código para a paralelização da função de distância

```
1  float distance(float* coord1_ptr, float* coord2_ptr) {
2
3        coordinate coord1 (coord1_ptr);
4        coordinate coord2 (coord2_ptr);
5        auto subs = coord2 - coord1;
6        auto pows = subs*subs;
7        scalar<float> s = reduce<plus<float>>(pows);
8        return sqrt(s.value());
9  }
```

A versão paralelizada da distância euclidiana encontra-se na listagem 4.6 assim como o respetivo esquema do funcionamento do código na figura 4.5. A função recebe as coordenadas dos dois pontos, armazenando-as em duas estruturas de dados de nome *Coordinate*, sendo este nome um alias definido no âmbito do trabalho para um *array* do *Marrow* (linhas 3 e 4). Os valores das coordenadas são subtraídos e multiplicados para obter o quadrado da distância em relação a cada um dos eixos x, y,e z (linhas 5 e 6). Sobre este valor é feita uma operação de redução onde se soma cada uma das distâncias dx^2, dy^2 e dz^2 de forma a obter a distância geral ao quadrado (d^2) representado por um escalar (linha 7). A operação termina com o retorno da raiz quadrada do resultado obtido previamente pela execução dos *kernels*.

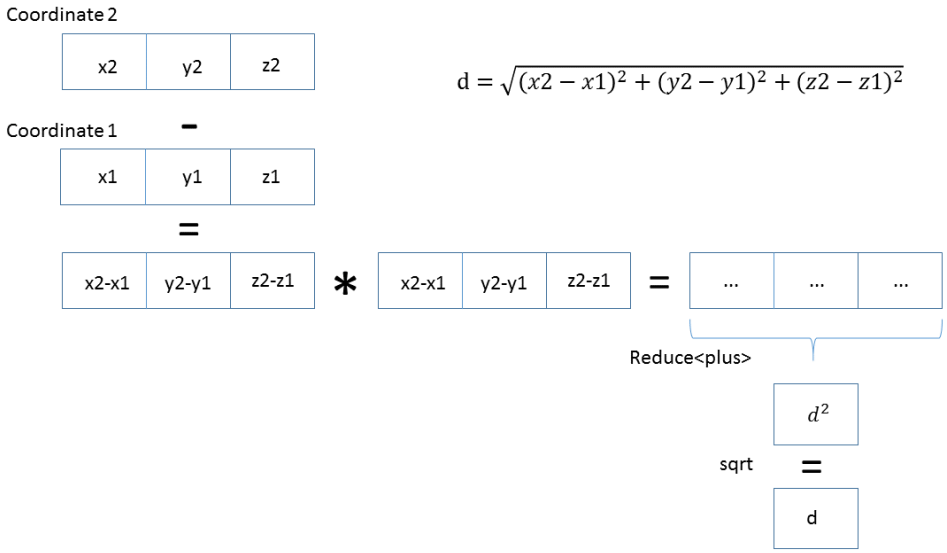

Figura 4.5: Esquema ilustrativo do funcionamento da função de distância euclidiana paralelizada.

4.4.3 Função *isInnerPoint*

Tendo em conta o que foi descrito na abordagem anterior e o comportamento do troço referido que executa a digitalização para grelhas tridimensionais (sub-secção 4.2.4), a paralelização da função de distância euclidiana apenas garante a aceleração do cálculo das coordenadas entre um par de pontos da grelha e um átomo de cada vez. No entanto para completar a etapa de construção da grelha base, para um *docking* entre proteínas de grandes dimensões, o algoritmo pode necessitar de efetuar milhões de cálculos de distâncias. Para estas circunstâncias, a quantidade de pontos da grelha para avaliar assim como de átomos é muito superior comparando com situações em que os intervenientes são proteínas pequenas. Como tal a solução não garante a escalabilidade com o tamanho das estruturas, pelo que foi necessário garantir que a solução efetua o maior número possível de cálculos e verificações sobre os pontos da grelha. Com esta solução pretendeu-se a paralelização da iteração que é feita sobre as coordenadas dos átomos da proteína, sendo esta iteração executada na função *isInnerPoint*. A otimização permite determinar a distância de um ponto da grelha a todos os átomos em simultâneo em alternativa a um átomo de cada vez.

Na listagem 4.7, encontra-se o código para uma função implementada no *Marrow* com intuito de melhorar o desempenho da função *isInnerPoint* sequencial. Com esta função implementada, o procedimento para a *isInnerPoint* deixa de ser feito para um átomo de cada vez mas sim para todos em simultâneo como pode ser verificado na figura 4.6.

A função recebe como argumentos a coleção de átomos (*FPoints*), a coleção de distâncias de VdW (*FRads*), as coordenadas do ponto corrente da grelha e o número total de átomos e distâncias de VdW a considerar. A coleção de pontos é armazenada numa matriz do *Marrow* e a de distâncias num vetor (linhas 8 e 10). Com estas duas estruturas é feita uma operação de subtração (linha 11). Tendo em conta que o termo mais à esquerda do RHS é uma matriz (*container* bi-dimensional) e o vetor é unidimensional, a operação tem como resultado uma matriz. Cada linha da matriz representa a subtração entre as coordenadas de um átomo e as do ponto. Esta matriz é multiplicada por ela própria de forma a obter as distâncias ao quadrado do ponto da grelha a todos os átomos da proteína para cada eixo cartesiano (linha 12).

A distância geral ao quadrado é obtida através de uma operação de redução-soma entre as colunas da matriz anterior, tendo como resultado um vetor de distâncias ao quadrado de tamanho $\#FPoints$ (linhas 14 e 15). Este vetor é depois usado para efetuar uma comparação com o quadrado das distâncias de Van der Waals usando a estrutura respetiva e respeitando a inequação $d^2 <= FRads^2$ que determina se o ponto é interno ou externo em relação a cada um dos átomos (linhas 17 e 18). O resultado é um *array* de booleanos ao qual é aplicado uma redução-Or de forma a produzir um escalar que assume o valor 1 se o ponto corrente da grelha for interno e 0 caso contrário (linhas 19 e 20).

Listagem 4.7: Código para a paralelização da função *isInnerPoint*

```
bool bigger::isInnerPointM(float* C, float fPoints[][number_dimensions]
, float* FRads, int fpointsNR, int fRadsNR){

    auto nd = number_dimensions;
    coordinate Ca(C);
    float (*ptr)[nd] = fPoints;
    auto *ptr2 = (float*)ptr;

    d_matrix<float,number_dimensions>fPointM(ptr2, nd*fpointsNR);
    vector<float> fRadsP(FRads,(size_t)fRadsNR);
    auto subs = (fPointM - Ca);
    auto pows = subs*subs;

    vector<float> d(fpointsNR);
    d = reduce<plus<float>>(pows);

    vector<bool> d2(fpointsNR);
    d2 = (d - pow(fRadsP,2)) < 0;
    scalar<bool> result = reduce<logical_or>(d2);

    return result.value();
}
```

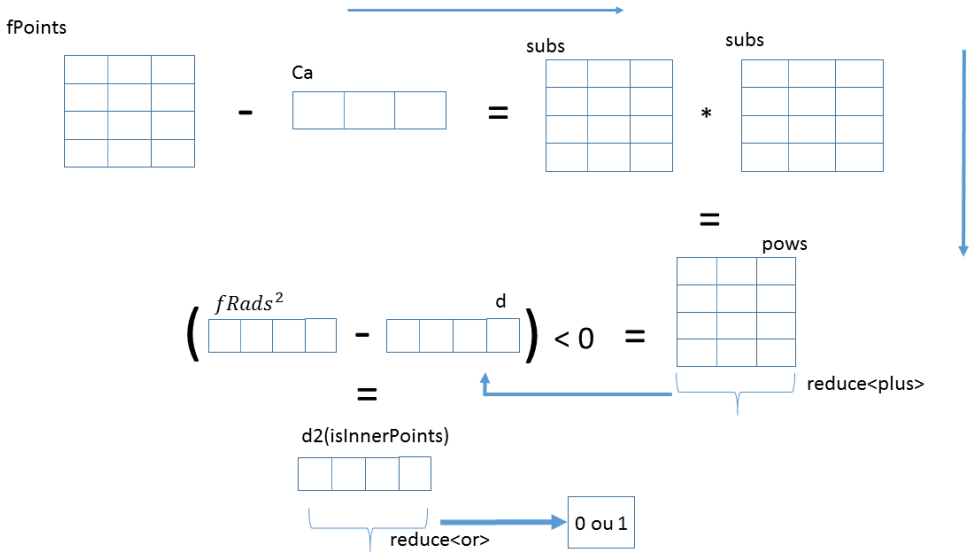

Figura 4.6: Esquema ilustrativo do funcionamento da função *isInnerPoint* paralelizada.

4.4.4 Etapa de digitalização para grelhas tridimensionais

A paralelização referida no ponto anterior permite acelerar o processo de determinação de pontos internos ou externos, melhorando o desempenho do BiGGER na etapa da digitalização para grelhas tridimensionais. No entanto tem uma falha: apesar de determinar as distâncias considerando todos os átomos em simultâneo, o processo continua a ser

aplicado a apenas um ponto da grelha de cada vez.

Na figura 4.7 é possível verificar o número de elementos de avaliação de distâncias assim como o número de elementos da grelha para avaliar se são internos ou externos. Os dados demonstram que o número de pontos da grelha é sempre superior ao número de átomos e que ambos aumentam com a complexidade dos cenários de *docking*, demonstrando a necessidade de otimizar a paralelização da função *isInnerPoint* para acelerar o processo de iteração dos pontos da grelha.

Esta otimização foi aplicada ao procedimento *BuildBaseGrid* (linhas 15-37 do pseudo-código 4.2) presente na unidade de *docking* que usa a função *isInnerPoint*: *linegrids*, sendo este o procedimento onde se encontra o fluxo principal para a digitalização das proteínas para grelhas tridimensionais. O código para a versão sequencial do procedimento contém três ciclos aninhados onde as coordenadas de cada ponto da grelha são convertidas com base na resolução da grelha (float com valor por omissão 1.0f). Com cada um dos pontos da grelha convertidos o procedimento chama a função *isInnerPoint*, contendo esta quatro ciclos adicionais.

A grelha é representada no código por um apontador de apontadores para linhas da grelha (linhas 17 e 18), sendo cada linha representada por apontadores para segmentos que se definem como *arrays* de duas posições (linha 16). A primeira posição do segmento indica onde este começa e a segunda onde termina. Para além destas estruturas existe um *array* dinâmico para a profundidade (eixo z) da grelha, chamado *zline*. Os valores binários respetivos à região interna/externa de cada uma das linhas são guardados no *zline*, sendo calculados no *BuildBaseGrid*. De seguida é executada uma função escreve o conteúdo do *zline* na linha corrente (*IntegersToLine*), passando o próprio como argumento (linha 34).

A nova versão permite a determinação de forma simultânea da região interna e externa para todos os pontos da grelha usando todos os pontos de avaliação. Uma das decisões de desenho de solução a considerar foi qual a estrutura de dados a usar para guardar os pontos da grelha, existindo duas possibilidades. Na figura 4.8 ilustram-se as alternativas assim como o resultado final de uma operação entre a matriz que guarda as coordenadas dos átomos e a estrutura que guarda os pontos da grelha. Foi concluído que optar por uma matriz para guardar os pontos da grelha faz com que o resultado final seja uma matriz de dimensões superiores às operandas. De notar que esta situação não permite o cumprimento da característica desejada para a otimização, da mesma forma para não existir perda de informação seria necessário que a matriz final tivesse dimensões superiores às operandas, não sendo uma opção realista.

Em alternativa foi optado por armazenar os pontos num *container* de nome *batch*, em que a respetiva indexação permite a subtração, para cada *worker* no *device*, das coordenadas de todos os átomos com as coordenadas de um ponto da grelha. A operação tem um *tensor* como resultado, pelo que esta alternativa tem como requisito a implementação de ambos *tensor* e *batch* como *containers* do *Marrow*.

A adição dos *containers tensor* e *batch* assim como a introdução de comportamentos dinâmicos nas estruturas *vector* e matriz será discutida mais adiante no capítulo 5.

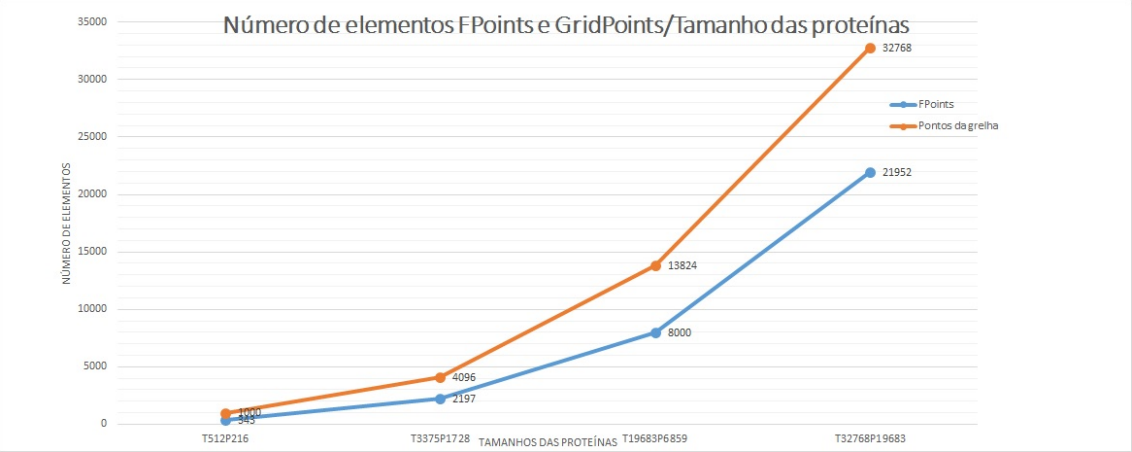

Figura 4.7: Gráfico linear associado ao número de coordenadas atómicas e pontos de grelha para cada um dos pares de proteínas usado nos testes.

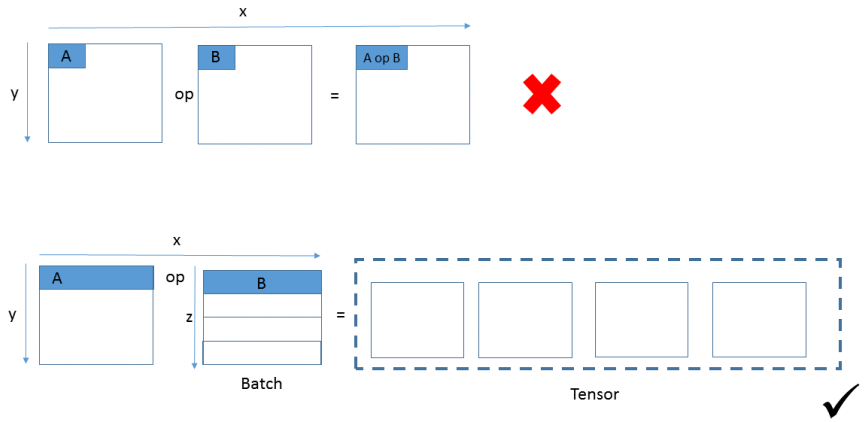

Figura 4.8: Esquema respetivo à abordagem usando uma matriz para os pontos da grelha vs usar *batch*.

4.4.4.1 Implementação da função *getZLine*

Tendo em conta os detalhes previamente referidos, esta otimização difere das restantes no sentido em que não é paralelizada uma função mas sim um conjunto de instruções do procedimento *BuildBaseGrid*. A função *getZLine* permite paralelizar a obtenção do *zline*, estando presente o código na listagem 4.8 e o raciocínio por detrás do código na figura 4.9. Esta função recebe como parâmetros o lado da grelha, o número de pontos de avaliação, as coordenadas e distâncias dos pontos de avaliação e um *array* onde estarão os valores binários para a grelha na sua integridade. A parte inicial da função é semelhante à função *isInnerPointM*, sendo efetuado a instanciação dos *containers* que irão guardar as distâncias e as coordenadas atómicas. De seguida é feita uma reconstrução da grelha no *Marrow_protein_docking* em alternativa a passar esta como parâmetro por ser a alternativa

mais simples.

Observando por impressão no terminal os valores das coordenadas para os pontos das grelhas permitiu concluir que variam entre $(0,0,0)$ e (x,y,z), sendo estes últimos valores o lado da grelha para cada um dos três eixos do referencial. Deste modo o *getZLine* está implementado para reconstruir a indexação da grelha através de 3 ciclos a variar entre 0 e $x/y/z$, sendo as coordenadas dos pontos desta adaptados para uma resolução pretendida e inseridas no *batch* recorrendo ao método *emplace*. Existe a possibilidade da presença dos ciclos na função ser prejudicial para o desempenho desta, o que será discutido no capítulo 7.

Após a conversão e inserção das coordenadas dos pontos da grelha no *batch* é executada de forma implícita a versão do *isInnerPointM*. Esta versão do *isInnerPoint* permite o cálculo das distâncias entre todos os pontos da grelha a todos os átomos da proteína que está a ser submetida ao passo de digitalização. Da operação de subtração entre uma matriz do *Marrow* onde estão guardadas as coordenadas atómicas e o *batch* com os pontos da grelha é obtido um *tensor* de ordem 3. O *tensor* é elevado ao quadrado para obter um *tensor* de distâncias quadráticas de cada eixo. Cada fatia deste *tensor* representa uma matriz de distâncias quadráticas de cada ponto da grelha a todos os átomos da proteína.

O *tensor* é submetido a uma operação de redução de três dimensões para duas, sendo a redução do tipo soma. A redução é aplicada às três distâncias quadráticas dos eixos, obtendo a distância quadrática geral. É gerada uma matriz cujo tamanho define-se como o produto do número de pontos da grelha por pontos de avaliação, sendo usada para a subtração dos seus valores com os quadrados das distâncias máximas aos pontos de avaliação para um ponto da grelha ser interno (*fRads*), verificando se é menor que 0.

Esta expressão gera uma matriz de booleanos (*isInnerPoints*) que indica se um ponto da grelha é interno em relação a um átomo da proteína, tendo as mesmas dimensões da matriz anterior. À matriz *isInnerPoints* é aplicada uma operação de redução<Or> a cada uma das suas linhas, sendo o resultado um *array* de tamanho número de pontos da grelha que permite guardar os valores binários de cada ponto da grelha no *array* que é passado como argumento.

Enquanto que na versão sequencial o *zline* continha apenas valores binários para uma linha da grelha, na versão paralelizada o *array zline* passa a conter valores binários para toda a grelha. Assim sendo no *BuildBaseGrid*, após o método *getZLine* ter sido executado, o *array* referido anteriormente é iterado de forma a executar a função *IntegersToLine* para um sub-conjunto de x elementos do *array*, sendo x o lado da grelha.

Listagem 4.8: Código para a paralelização da fase de construção da grelha base

```
void bigger::getZLine(int zline[], float FResolution
          , float fPoints[][number_dimensions], float FRads[]
          , int zlineElems, int fpointsNR, int fRadsNR){
      float halfres = 0.5f*FResolution;
      float (*ptr)[number_dimensions] = fPoints;
      auto *ptr2 = (float*)ptr;

      d_matrix<float,number_dimensions>fPointM(ptr2,fpointsNR);
       int nSegments = pow(zlineElems,3);
      vector<float> fRadsV(FRads,fRadsNR);

      d_batch<array<float,number_dimensions>>gridLineB(nSegments);
      //T1: Converts gridLinePoints to float coordinates
      // and inserts them into the batch
      int index = 0;
      for(auto z = 0; z < zlineElems; z++){
          auto zzline = z * FResolution + halfres;
          for(auto y = 0; y < zlineElems; y++) {
              auto yGridLine = y * FResolution + halfres;
              for(auto x = 0; x < zlineElems; x++) {
                  auto xGridLine = x * FResolution + halfres;
                  gridLineB.emplace(index
                  , std::initializer_list<float>{xGridLine
                  ,yGridLine,zzline});
                  index++;
              }
          }
      }
      //T2: calculate distances for all grid points
      auto subs = fPointM-gridLineB;
      vector<float,3> res (nSegments,fpointsNR,number_dimensions);
      res= subs*subs;
      int size = pow(zlineElems,3);
      vector<float,2> sumD(size, fpointsNR);
      sumD = reduce<plus<float>>(res);
      //T3: Determination for all grid points if they are internal ou not,
      //taking into accout all fPoints
      vector<bool,2> isInnerPoints(size, fpointsNR);
      isInnerPoints = (sumD - pow(fRadsV,2)) < 0;
      vector<int> toRet(zline,nSegments);
      toRet = reduce<logical_or>(isInnerPoints);
  }
```

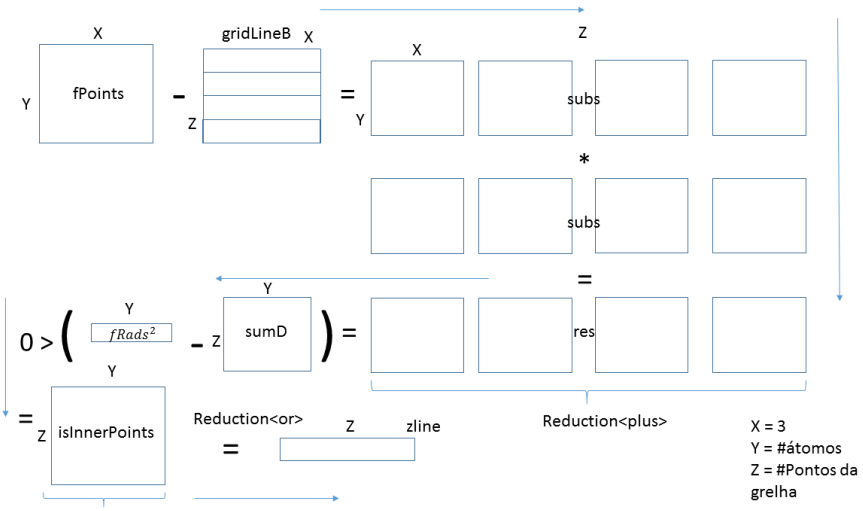

Figura 4.9: Esquema ilustrativo do procedimento para obter as regiões interna e externa da grelha base a partir do *Marrow*, usando a função *getZLine*.

4.4.4.2 *getZLine* particionado

A otimização referida no ponto anterior permite melhorar o desempenho do BiGGER, determinando as regiões interna e externa para todos os segmentos da grelha base em simultâneo. No entanto por observação dos requisitos teóricos de memória necessária para a alocação de cada um dos *containers* (tabela 4.2), pode-se verificar que existe uma falha a resolver. A memória necessária para alocar todas as estruturas é crescente com as dimensões da grelha base e com o número de átomos, sendo possível o *device* necessitar de valores de memória muito superiores ao que dispõe. Neste contexto, a possibilidade de ocorrerem erros de execução [16] relacionados com a falta de recursos no *device* é considerável. Tendo em conta esta possibilidade, foi necessário introduzir melhorias à solução atual de forma a diminuir estes requisitos.

A melhoria principal consiste em dividir a obtenção da grelha base em partições cujo número dependerá da memória livre presente no *device*. O primeiro passo consiste em determinar ambas memória necessária para alocar as estruturas e memória livre no *device*, sendo esta última subtraída pela memória necessária para as estruturas (1) e (2) da tabela 4.2. Na figura 4.10 ilustra-se as estruturas que fazem parte da solução *getZLine* agrupadas em duas categorias: particionáveis e não-particionáveis. De notar que as estruturas não-particionadas não podem ser divididas pois todas as coordenadas atómicas devem estar disponíveis na determinação das regiões interna/externa da grelha base. As restantes estruturas presentes na tabela podem ser particionadas, permitindo reduzir os requisitos de memória livre no *device* pois o número de segmentos a considerar é menor, mantendo o número de coordenadas atómicas constante.

O número de partições é determinado dividindo o valor de memória necessária para a alocação dos *containers* pelo mesmo valor para a memória livre no *device*. Se o número

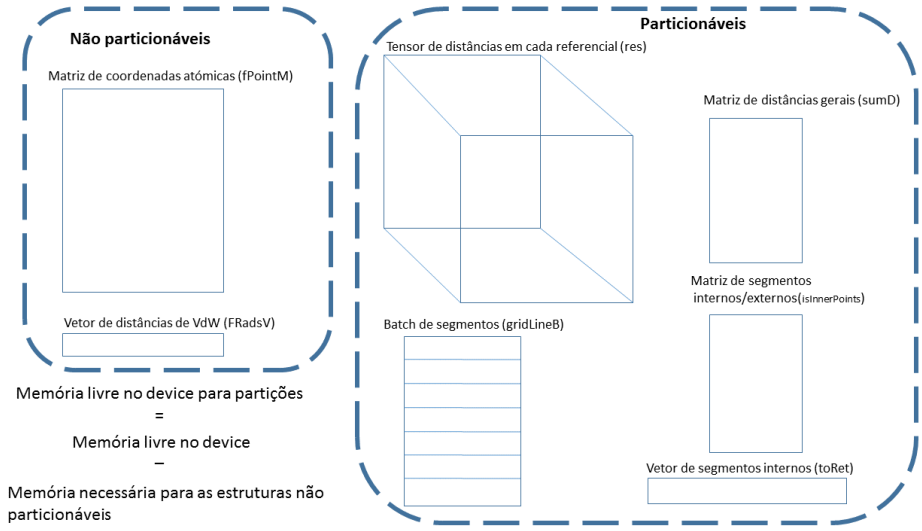

Figura 4.10: Estruturas particionáveis e não-particionáveis.

de partições a considerar for nulo existe memória suficiente no *device* e a computação não será particionada, caso contrário o *device* não tem memória suficiente e a computação terá de ser particionada. Nesta situação é determinado se as partições terão um número de elementos equivalente através do cálculo do resto da divisão inteira para a divisão anteriormente referida.

O lado da grelha para cada uma das partições é determinado assim como o número de segmentos que cada uma das partições tem. Com estes valores obtidos, versões reduzidas dos *containers* são criadas em alternativa, existindo um processo cíclico de chamadas a uma função auxiliar do *getZLine* para cada partição. Esta função auxiliar implementa os procedimentos referidos na subsubsecção 4.9 de forma a eliminar redundâncias de código.

A versão reduzida do vetor de segmentos processados para retornar ao BIGGER será utilizada para copiar os resultados de cada partição no correspondente com as dimensões originais. Na eventualidade de existirem segmentos por processar que não couberam nas partições, é feita uma última chamada para esta função utilizando *containers* com dimensões adequadas ao processamento dos segmentos restantes.

No final do procedimento existe a possibilidade de uma percentagem ínfima dos segmentos finais não terem sido processados, sendo um aspeto menos positivo da solução. Para esta situação o BiGGER foi alterado de forma a considerar estes segmentos como externos, existindo duas razões para esta opção:

- Estes segmentos estão localizados num dos 8 cantos da grelha base. Tendo em conta que o BiGGER ajusta as dimensões da grelha base em função do tamanho da proteína e centraliza esta com a grelha, existirá sempre uma região externa que se encontra nos seus limites.

- Os segmentos devem ser considerados externos de forma a que segmentos vizinhos

determinados como internos possam ser considerados como fronteira, na fase de determinação das grelhas central-superfície.

A segunda otimização introduzida consistiu em alocar as estruturas necessárias apenas uma vez em cada rotação iterada. A cada iteração das partições os *containers* particionáveis são reutilizados.

Estruturas de dados dinâmicas	**Memória necessária (Bytes)**
(1)Matriz de coordenadas de átomos	TF * numDimensões * numÁtomos
(2)Vetor de distâncias de cada um dos átomos	TF * numÁtomos
(3)*batch* com segmentos da grelha	TF * numDimensões * numSegmentos
(4)*tensor* de distâncias euclidianas por referencial	TF * numDimensões * numÁtomos * numSegmentos
(5)Matriz de distâncias euclidianas gerais	TF * numÁtomos * numSegmentos
(6) Matriz de regiões internas/externas para cada segmento	TB * numÁtomos * numSegmentos
(7)Vetor resultado	TI * numSegmentos

Tabela 4.2: Tabela com os requisitos de memória no *device* para cada um dos *containers* utilizados. TF, TB e TI são os tamanhos de um *float*, *boolean* e inteiro em Bytes, respetivamente. Por sua vez numÁtomos, numDimensões e numSegmentos indicam respetivamente o número de átomos (FPoints), número de dimensões do problema (3 dimensões) e o número de segmentos da grelha.

4.5 Sumário

Neste capítulo foram discutidas as questões relacionadas com a deteção dos problemas de desempenho do BiGGER, verificação sobre a possibilidade de serem paralelizadas e implementação das soluções. No total foram consideradas três soluções incrementais, sendo a versão inicial a que apenas paraleliza a função de distância euclidiana (*distance*), onde o *profiler* detetou que a execução passa mais tempo. A segunda solução apenas paraleliza a função que invoca a *distance* para determinar se um ponto da grelha é interno ou não (*isInnerPoint*). A terceira solução e final afeta a etapa de digitalização para grelhas tridimensionais onde é paralelizada a fase de construção da grelha base. A construção da grelha base consiste em um procedimento onde cada ponto é submetido a uma averiguação se é interno ou não dependendo da sua distância aos átomos da proteína que está a ser digitalizada (*BuildBaseGrid*). No capítulo seguinte serão discutidas as contribuições no *Marrow* de forma a suportar a solução *getZLine*.

Capítulo 5

Contribuições no *Marrow* para suportar a solução final

Na última parte do capítulo 4 a implementação da solução final, a versão *getZLine*, foi discutida. Serve o presente capítulo para relatar detalhes de implementação das componentes necessárias para o bom funcionamento da solução que o *Marrow* anteriormente não dispunha.

5.1 *tensors*

Um *tensor* é um *array* multidimensional, podendo o número de dimensões ser variável. Kolda e Balder no seu artigo sobre decomposição de *tensors* [19] definem um *tensor* de ordem N como o produto de N espaços vetoriais, cada um com o seu sistema de coordenadas. Para estas estruturas existe a noção de ordem, sendo esta o número de dimensões que o *tensor* dispõe para representar os seus elementos. Neste contexto um *tensor* de ordem 0 representam escalares, com ordem 1 representam vetores e com ordem 2 matrizes. *tensors* de ordem 3 ou superior denominam-se *tensors* de ordem superior. Uma segunda noção a considerar é o *rank* de um *tensor* X, sendo este um indicador para o número mínimo de *tensors* de ordem 1 necessários para formar X por soma.

Com a adição desta estrutura como *container* do *Marrow*, tornou-se possível usar a *framework* para implementar *kernels* que paralelizam problemas com um número de dimensões superior a dois. Em termos de operações, o *tensor* tem as mesmas que os restantes *containers* enquanto que a maneira de instanciar um *tensor* difere. Para instanciar um *tensor* é necessário para a versão estática passar como argumento de *template* o tipo dos elementos a guardar assim como um conjunto de dimensões *dims*. Cada um dos elementos do *dims* é referente ao limite de elementos a guardar no *tensor* para a dimensão respetiva. Um exemplo para elucidação: considere-se um *tensor* instanciado usando $dims < 3, 4, 5 >$.

Nesta situação o *tensor* será tridimensional e terá 3 fatias de tamanho 4x5.

Estas estruturas são adequadas para a resolução do problema indicado na presente tese pois permitem agregar a informação relacionada com os pontos da grelha. No contexto da resolução do problema recorreu-se a um *tensor* para guardar as distâncias ao quadrado Dx,Dy e Dz de cada um dos pontos da grelha a todos os pontos da coleção de coordenadas atómicas, referentes a cada um dos três eixos cartesianos. A distância geral de cada ponto da grelha é depois obtida através da contração do *tensor* para uma matriz de distâncias via operação de redução. Sobre os *tensors* de ordem 3, o artigo refere os conceitos geométricos de fatia e fibra. Uma fatia consiste numa secção bi-dimensional do *tensor* e uma fibra uma linha/coluna/tubo deste. No caso do *tensor* de distâncias, este terá tantas fatias como pontos da grelha, em que cada uma destas fatias tem de dimensões número de coordenadas atómicas por 3, existindo no total tantas fatias como pontos da grelha.

5.2 *batch*

O *batch* é uma estrutura que permite a definição de um *container* de dimensão N que contém *sub-containers* de dimensão $N-1$. Esta característica permite formar *tensors* de ordem 3 quando um *batch* é invocado como termo em conjunção com uma *container* bidimensional nas operações do *Marrow*.

Um dos desafios encontrados na implementação do *batch* no *Marrow* consistiu em determinar um esquema de indexação para as *threads* poderem aceder aos seus elementos. Quando uma matriz é invocada para uma operação, o *Marrow* inclui no código OpenCL gerado uma variável que representa um índice usável por parte das *threads* para aceder ao conteúdo da matriz. Esta variável tem o nome de *index01*, também referível por *indexXY* pois o seu valor depende das posições x e y do elemento a aceder assim como do número de colunas da matriz. Quando uma expressão tem como resultado um *tensor* de ordem 3, é necessário declarar outra variável índice que permita indexar todos os elementos distribuídos pelas matrizes do *tensor*. Desta forma o esquema de indexação passa a incluir o uso do *index01* assim como o número de matriz do *tensor* e o número de linhas de cada matriz, sendo este uma variável de nome *index012* (ou *indexXYZ*).

No entanto o *Marrow* não suportava o esquema de indexação para o *batch*, sendo que neste caso o código OpenCL deve incluir uma variável permita indexar elementos dentro de elementos. De notar que o *batch* assume a particularidade de os seus elementos poderem ter dimensões superiores a 0, ou seja, os elementos do *batch* são sub-coleções de elementos.

A solução consistiu em garantir que o *Marrow* declare uma variável *index02* (ou *indexXZ*) no momento em que percebe que uma expressão contém *batches*. O valor do *index02* depende do número de sub-coleção de elementos z, da posição x dos seus elementos e do número de elementos de cada sub-coleção. A componente z do *index02* é coincidente com o número de matriz referido no esquema de indexação para o *tensor*.

Na figura 5.1 ilustra-se uma demonstração dos cálculos dos índices para os elementos nos três *containers*. Considera-se uma *thread* de um conjunto de 27 *threads* lançadas que devem desempenhar uma operação de soma entre uma matriz e um *batch*, resultando num *tensor* de ordem 3. A *thread* assume a responsabilidade de somar o elemento da matriz $(x, y) = (1, 2)$ com o elemento da primeira sub-coleção do *batch* $(x, z) = (1, 0)$. Tendo em conta o esquema de indexação o elemento na posição $(x, y, z) = (1, 2, 0)$ do *tensor* (elemento para o qual $index123 = 7$) terá o resultado da soma. Com esta demonstração garante-se que a validação do esquema de indexação para os três elementos.

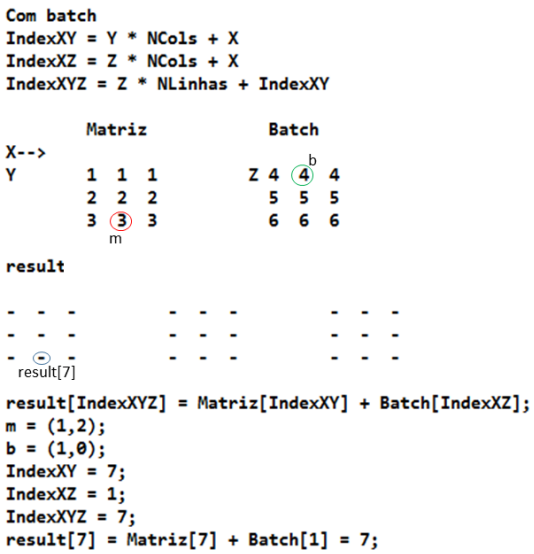

Figura 5.1: Demonstração dos cálculos para a indexação dos elementos do *tensor*.

Outro problema encontrado na referida implementação tem a ver com a transferência de memória para o *device*. Um dos requisitos do *Marrow* sobre as zonas de memória dos elementos a guardar nos *containers* indica que estas devem ser contíguas. No caso do *batch* este requisito não é cumprido, pois num dos testes implementados para o *batch* foi verificado que a zona de memória do conteúdo a introduzir no *batch* não era contígua. Por consequência o teste não passava devido à computação de dados com valor incorreto.

A solução passou por adicionar uma propriedade a todos os *containers* que permite identificar se são contíguos. Na eventualidade de não serem, a transferência de memória passa a ser efetuada por um novo mecanismo que permite resolver transferências de dados de *containers* não contíguos que antes da elaboração da tese não era suportado pelo *Marrow*. Para efetuar uma transferência de dados em zona de memória contígua, o *Marrow* efetua uma única cópia abrangente. Por sua vez na cópia de dados não contíguos, as transferências são executadas em blocos de tamanho fixo, dependente do número de elementos e do tipo em cada linha do *batch*. Cada bloco corresponde a uma operação de cópia, permitindo transferir os elementos não contíguos do *host* para um bloco de

memória contíguo no *device*.

O papel do *batch* na solução corrente consiste em guardar os triplos de índices dos pontos da grelha $(i1, i2, i3)$ convertidos, sendo o *batch* declarado numa subtração com a matriz onde os pontos de avaliação estão guardados de forma a obter um *tensor* de distâncias ao quadrado (dx^2, dy^2, dz^2). De forma a armazenar os pontos no *batch* foi implementada uma função de nome *emplace*. Esta função recebe como argumentos os elementos necessários para passar a um construtor do tipo de elementos do *batch* e a posição do futuro elemento.

5.3 Dinamização das estruturas atuais

Como referido previamente, usar os *containers* do *Marrow* em condições estáticas seria contraproducente para a resolução da tese. Foi optado por introduzir novas versões para todos os *containers* usados nesta versão, mais precisamente adicionar dois tipos de construtores: (1) Construtores que recebam o número de elementos a guardar; (2) Construtores que recebam para além do número de elementos, o conteúdo destes passando um apontador como argumento. Estes construtores permitem a construção das estruturas sem ter a necessidade de passar o tamanho como argumentos de *template*. Adicionalmente permitem criar as estruturas e guardar o conteúdo dos elementos sem recorrer às operações de cópia do C++ [51] que de acordo com a complexidade temporal apresentada (O($last - first$)) podem ser degradantes para o desempenho da solução. De notar que apenas serão detalhadas as implementações efetuadas nos *containers*. Para além destas, foi necessário efetuar alterações na componente de *runtime* da *Marrow* que não serão consideradas no presente documento. Estas alterações estão fora das contribuições indicadas para a dissertação, sendo confiado ao Professor Doutor Hervé Paulino a sua aplicação.

5.3.1 Matrizes

No caso das matrizes, foi necessário adicionar dois tipos novos de matrizes: (1) matrizes com número de linhas dinâmico e (2) matrizes com ambos número de linhas e colunas dinâmicos. Estas últimas diferem da matriz estática original no sentido de os elementos serem agrupados por vetores em vez de *arrays*. A diferença entre um vetor e um *array* resume-se ao primeiro ser uma estrutura dinâmica, isto é, o seu tamanho depende do número de elementos guardados em tempo de execução, o *array* por sua vez é uma estrutura estática.

A implementação para o segundo tipo de matrizes identificado passou por modificar o vetor de forma a suportar mais do que uma dimensão. Para indicar o número de dimensões de um vetor deve ser passado o valor como argumento de *template*, caso contrário por omissão é sempre unidimensional. Um operador *get* foi implementado para vetor, passando este a ter um comportamento similar a uma matriz em situações com o número de dimensões do vetor superior ou igual a 2. Este operador retorna um sub-vetor para um

dado índice de linha em vez de um *array*, sendo o sub-vetor em questão um vetor com $numeroDimsVetorPai - 1$ dimensões.

5.3.2 *tensor* e *batch*

Tendo em conta o facto de os vetores suportarem mais do que uma dimensão, a implementação foi aproveitada para dinamizar os *tensors*. A solução atual para os vetores não tem em conta a característica dos *tensors* poderem suportar um valor arbitrário de dimensões, sendo necessário implementar construtores que recebam os limites para cada uma das dimensões do vetor como argumento.

Para o problema indicado na tese o requisito imposto para os *tensors* é serem de ordem 3, como tal foi implementado um construtor que suporta a instanciação de vetores tridimensionais, sendo estes usados no lugar dos *tensors* para a solução dinâmica.

Para o *batch*, a sua implementação original foi estendida adicionando um construtor especifico para a situação em que não é conhecido o número de linhas em tempo de compilação. De forma a que o compilador consiga diferenciar ambos, a técnica de "*Substitution failure is not an error*" (SFINAE) [47] foi aplicada.

5.4 Sumário

Foram discutidas as inclusões necessárias para o *Marrow* conseguir suportar a solução final ao problema da presente dissertação. Com a adição dos *containers tensor* e *batch* passa a ser possível utilizar a ferramenta para problemas com dimensões superiores a 2. Por sua vez a alteração dos *containers* existentes para suportar comportamentos dinâmicos permite ao *Marrow* a sua utilização em problemas onde é desconhecido o número de elementos.

Capítulo 6

Avaliação de desempenho

Este capítulo destina-se à avaliação do desempenho da nova versão do algoritmo BiGGER, tendo em conta os detalhes de implementação referidos nos dois capítulos anteriores. São abordadas questões como a metodologia de avaliação do trabalho efetuado, os parâmetros e circunstâncias de teste assim como o equipamento utilizado na execução. No final é apresentada uma reflexão sobre os resultados.

6.1 Metodologia de avaliação

O BiGGER paralelizado foi submetido a uma bateria de testes em que foram utilizadas interações realistas. As interações foram selecionadas recorrendo a uma *benchmark*[7] da Zlab [58]. Em termos de validação, foram implementados testes unitários para as três soluções abordadas no capítulo anterior assim como para os elementos que estas necessitam.

6.1.1 Parâmetros de teste

A página da *benchmark* referida anteriormente contém uma tabela com 231 interações adequadas e agrupadas por três categorias: (1) interações *rigid-body*, (2) dificuldade média, (3) difícil . Os tuplos desta tabela indicam os códigos PDB das proteínas intervenientes assim como os nomes e a métrica de avaliação respetiva (RSMD). Foi optado pelas interações cujo valor para a métrica RSMD é o máximo de cada um dos agrupamentos, sem nenhuma razão a apontar pois o objetivo da tese foca-se no aumento do desempenho e não na validação dos resultados da versão paralelizada. No entanto escolher interações de cada um dos agrupamentos permitiu avaliar o comportamento das versões paralelizadas em função da complexidade de cada um dos cenários.

[7] Uma *benchmark* é uma coleção transferível de ficheiros de dados (PDB) de proteínas frequentemente utilizadas em testes de *docking*

Adicionalmente foi escolhido um dos testes do agrupamento difícil cujo número rotações seja próximo de 15000, sendo este o valor máximo utilizado nos testes com proteínas fictícias. De notar que este valor é determinado automaticamente pela componente *dockprep*.

Desta forma na tabela 6.1 estão indicados os três cenários de teste escolhidos, sendo a interação I1 a menos complexa do conjunto e a I3 a mais complexa. De notar que as colunas "Recetor" e "Ligando" têm como valores o código dos ficheiros .pdb a transferir da base de dados de proteínas RCSB PDB [5].

	Recetor	#ÁtomosR	Ligando	#ÁtomosL	Rotações*
I1 (rigid-body)	3CHY	1166	1FWP	1057	107
I2 (difícil)	1G0Y_R	2694	1ILR_1	2425	3782
I3 (difícil)	1FAK_HL	4739	1TFH_B	3081	17831

Tabela 6.1: Interações de teste escolhidas
*Para o caso das rotações, o seu valor foi determinado pela componente *dockprep* do Open-Chemera.

6.1.1.1 Sobre as interações escolhidas

A situação de teste I1 contextualiza uma interação relacionada com bactérias, precisamente a E.Coli (*Escherichia coli*). A bactéria E.Coli encontra-se presente no nosso sistema digestivo, podendo as estirpes mais perigosas causar as doenças digestivas mais comuns. Julius Adler (1966)[1] determinou que a E.Coli sofre das propriedades da quimiotaxia (Chemotaxis)[2], atraindo-se por fontes de oxigénio assim como de energia como por exemplo a glucose. O estudo da quimiotaxia associada à E.Coli é o foco da situação I1, em que uma proteína CheY com o papel de recetora interage com uma CheA [11].

A situação de teste I2 diz respeito ao confronto entre dois pares de interleucinas[1] do tipo IL-1. Segundo Herman Schreuder et al (1997) [45] esta interação é o resultado inicial da reação do nosso organismo a uma inflamação causada por infeção ou danos no tecido. Uma interleucina IL-1 assume o papel de recetora enquanto que a versão antagonista (IL1RA) o de ligando.

A interação mais complexa da bateria de testes (I3) envolve dois fatores: um fator de coagulação sanguínea do tipo VII (recetor) e um fator tissular solúvel (ligando). A associação entre ambas as proteínas permite iniciar o processo de coagulação do sangue. Esta interação é abordada em detalhe no artigo de Erli Zhang, Robert St. Charles e Alexander Tulinsky (1999) [61].

6.1.2 Equipamento utilizado

Para executar os testes recorreu-se a um *cluster* suportado pelo DI-FCT/UNL composto por quatro nós de computação com capacidades de GPGPU e um nó principal responsável

[2] A quimiotaxia é o nome dado ao movimento de um organismo em resposta a estímulos químicos[44].
[1] Interleucinas são proteínas derivadas dos leucócitos, ou glóbulos brancos.

por gerir a computação entre os anteriores. Cada nó contém uma CPU Intel Core i7-3930K e duas GPU da Nvidia: uma GeForce GTX 1080 Ti e uma GeForce GTX TITAN X ou GeForce TITAN Xp. As características do *cluster* encontram-se sintetizadas na tabela 6.2.

CPU	RAM
Intel Core i7-3930K Processor	64GB
GPU1	Nvidia GeForce GTX 1080 Ti 11GB
GPU2	Nvidia GeForce TITAN Xp 12GB
GPU3	Nvidia GeForce GTX TITAN X 12GB

Tabela 6.2: Características do equipamento presente no *cluster* a ser usado nos testes.

6.2 Validação do trabalho efetuado

A validação das funções implementadas na biblioteca *Marrow* (*isInnerPointM* e *getZLine*) assim como das implementações intrínsecas do *Marrow* foi verificada recorrendo à *framework* dedicada a testes da Google: GTest, sendo cobertas situações específicas. No caso dos testes para as implementações intrínsecas, estes têm o intuito de validar:

- A criação de *containers*;
- Manipulação dos seus elementos por expressões do *Marrow*;
- A geração correta do código para os *kernels* OpenCL;
- Transformação recorrendo aos esqueletos usados para a solução;
- Comprovação de características presentes nos *containers*;

6.2.1 Testes para as soluções *isInnerPointM* e *getZLine*

Testes unitários para ambas funções *isInnerPointM* e *getZLine* foram implementados, tendo o fundamento de garantir que não são levantados erros de execução do OpenCL para os *kernels* criados por esta função. Os parâmetros de teste (número de átomos a considerar e lado da grelha tridimensional) devem ser grandes o suficiente para o *Marrow* considerar a execução com a GPU em alternativa à CPU. De notar que atualmente as reduções no *Marrow* usando a CPU são instáveis, levando a erros de execução. O objetivo destes testes consiste em avaliar se todas as operações terminam, não considerando relevante a validação. Esta última é avaliada nos testes unitários do *Marrow* para cada uma das operações. Como tal foram considerados os testes:

- IsInnerPointGPU : O único teste para a solução *isInnerPointM*, usando 2197 átomos para determinar o estado interno de uma célula da grelha.
- IsInnerPointGridLine44E: Teste unitário para a interação I1;
- IsInnerPointGridLine71: Teste unitário para as interações I2 e I3;

6.3 Resultados

Os resultados que a seguir serão apresentados dizem respeito aos ganhos de desempenho que são obtidos, estando divididos em dois agrupamentos. O primeiro agrupamento contém os resultados obtidos pela simulação do procedimento de cálculo dos pontos internos com e sem o *batch*. Foi usada a *Marrow-benchmark* [26] para este efeito, sendo que os resultados servem para comprovar que o uso do *batch* é benéfico para o desempenho do BoGIE. São testadas as versões em que passamos a média de átomos utilizados em cada segmento e em que passamos a totalidade dos átomos. Os resultados avaliam especificamente o tempo de inicialização do *batch*, onde são guardadas as coordenadas dos segmentos. Adicionalmente são avaliados os tempos de computação para a etapa da solução *getZLine* onde é utilizado o *batch*, confrontando com os tempos para a mesma etapa na versão sequencial. Os testes foram executados usando 20 partições para o *batch*.

Na figura 6.1 podem ser consultados os *containers* utilizados para simular o procedimento para ambas as versões. A simulação da versão sequencial do procedimento onde é utilizado o *batch* é feita usando um *array* onde são guardadas as coordenadas de um segmento de cada vez, que será usado para subtrair com uma matriz com as coordenadas atómicas, resultando numa outra matriz que multiplicando por ela própria obtém-se as distâncias em cada eixo de referência do segmento a todos os átomos usados. A interação I1 e não pôde ser avaliada devido ao excesso de tempo de execução, derivado da quantidade de tarefas que a *Marrow* lança. Este fator faz com que existam demasiadas tarefas (*tasks*) para o OpenCL conseguir lidar, impossibilitando a obtenção de resultados.

O segundo agrupamento contém os resultados respetivos à execução parcial de ambas solução *getZLine* e sequencial. De notar que não foi possível a execução total do BiGGER com a integração da *Marrow* pelos motivos que serão relatados mais adiante no capitulo 7. No entanto esta versão consegue executar uma rotação, pelo que se optou por determinar os resultados por uma execução parcial. De notar que no segundo agrupamento os testes são efetuadas com um escopo de procura de átomos que considera a grelha toda e não uma porção. Sendo que estes resultados têm o objetivo de demonstrar que apesar da solução *getZLine* ainda não ter o desempenho ideal está no bom caminho, comprovando o bom funcionamento dos elementos implementados para esta solução.

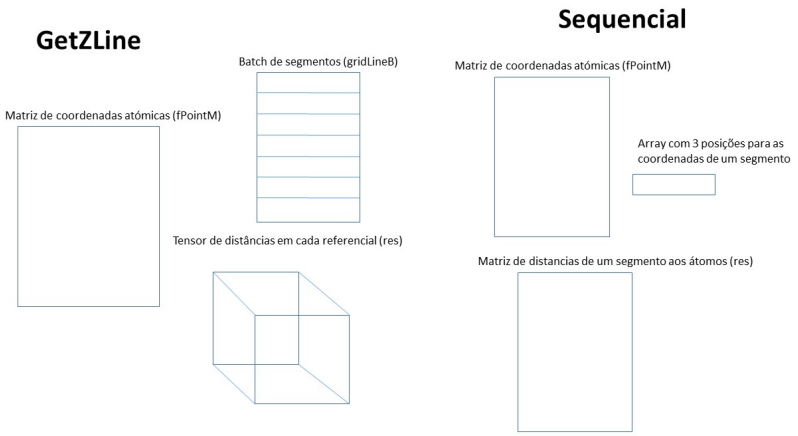

Figura 6.1: *containers* utilizados para simular as computações do *batch*

6.3.1 Tempos de execução

6.3.1.1 Execução parcial da digitalização para grelhas tridimensionais em cada uma das interações

	Tempos de digitalização parciais (s)	
	getZLine	Sequencial
I1	6	13
I2	10	100
I3	200	3600

Tabela 6.3: Tempos de computação para a etapa de digitalização de grelhas tridimensionais quando a totalidade dos átomos são usados. Os tempos são respetivos à versão sequencial e à solução *getZLine*.

6.3.1.2 Execução usando a média de átomos usados por segmento

	I1	I2	I3
Tempo médio de inicialização	-	26	26
Tempo médio de computação	-	66	52

Tabela 6.4: Tempos médios simulados de inicialização do *batch* e computação paralelizada (em ms).

6.3.1.3 Execução usando a totalidade dos átomos

	Tempos de computação (ms)	
	Com *batch*	Sequencial
I1	-	-
I2	66	16933
I3	52	12574

Tabela 6.5: Tempos de computação da solução com *batch* assim como da versão sequencial

	I1	I2	I3
Tempo médio de inicialização	-	26	25
Tempo médio de computação	-	2027	2192

Tabela 6.6: Tempos médios simulados de inicialização do *batch* e computação paralelizada (em ms).

	Tempos de computação (ms)	
	Com *batch*	Sequencial
I1	-	-
I2	2027	19231
I3	2192	14448

Tabela 6.7: Tempos de computação da solução com *batch* e sem *batch* quando a totalidade dos átomos são usados.

6.3.2 Ganhos de desempenho para cada uma das interações

6.3.2.1 Uso do *batch* na solução *getZLine*

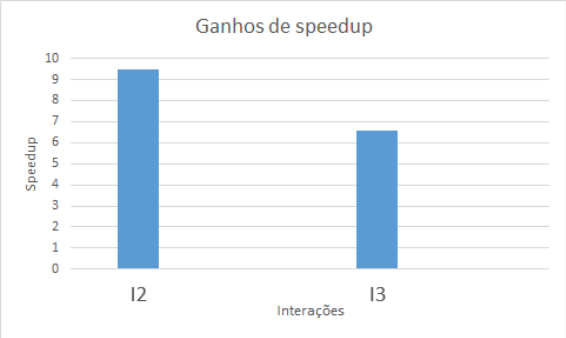

Figura 6.2: *speedup* médio quando é utilizada a totalidade dos átomos no procedimento.

6.3.2.2 Digitalização em grelhas tridimensionais para cada uma das interações

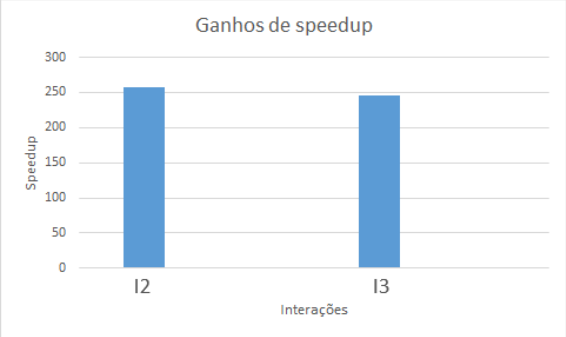

Figura 6.3: *speedup* médio quando é utilizada a média de átomos necessários para cada segmento.

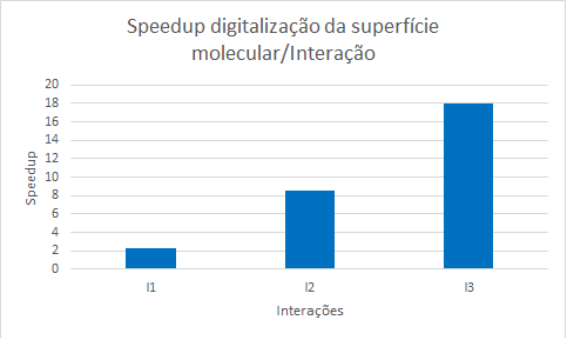

Figura 6.4: Ganhos de desempenho obtidos pela execução parcial da digitalização para grelhas tridimensionais de ambas solução *getZLine* e programa sequencial em cada uma das interações.

6.3.3 Análise dos resultados obtidos

Por análise aos resultados é verificável o crescimento do *speedup* à medida que a complexidade das interações aumenta. Apesar de serem necessários três ciclos para inicializar o *batch*, esta etapa não revelou ser prejudicial para o desempenho da solução pois demora menos do que o passo de computação do *tensor* de distâncias em todos os testes como se ilustra nas tabelas 6.4 e 6.6. Sobre os tempos de computação, de notar que a versão paralelizada do procedimento demorou menos tempo do que a sequencial como pode ser ilustrado nas tabelas 6.5 e 6.7. Os ganhos de desempenho (figuras 6.3 e 6.2) são mais acentuados quando é passada a média de átomos usados por cada segmento em vez da totalidade.

Em termos de desempenho da fase de digitalização de referir que o *speedup* é crescente com a complexidade das interações. No caso da interação I3 a versão sequencial demorou demasiado tempo a executar cada uma das rotações. Tendo em conta que ao fim de uma hora o programa não tinha completado uma rotação considerou-se que o *speedup* para esta interação é pelo menos 18 em relação ao BiGGER sequencial, podendo ser bastante superior.

Capítulo 7

Conclusões

7.1 Contribuições

7.1.1 *Marrow*

Apesar de ser considerado uma ferramenta vantajosa, o *Marrow* na versão antecedente ao trabalho efetuado não reunia as condições necessárias para ser usado como ferramenta na resolução do problema. Consequentemente, ocorreram demasiados contra-tempos que revelaram ser difíceis de resolver. Destes, o mais grave foi o facto de os *containers* do *Marrow* estarem preparados apenas para problemas computacionais de tamanho fixo. Outro exemplo a considerar foi a falta de estruturas adequadas para armazenar o conteúdo das grelhas tridimensionais, sendo que apenas existiam na versão antecedente *containers* com até duas dimensões. A resolução destes imprevistos representou uma oportunidade de crescimento da *framework*, passando a estar preparado para integração em soluções de projetos futuros que resolvam problemas computacionais com características semelhantes.

7.1.2 Open-Chemera

Independentemente do desempenho da versão paralelizada do BiGGER, destaca-se a inclusão de novas ferramentas para efetuar *profiling* ao algoritmo, sendo esta uma contribuição suplementar. Qualquer futuro programador com a tarefa de otimizar o desempenho do BiGGER poderá utilizar ou até mesmo melhorar os três *scripts* implementados para facilitar a tarefa de executar e obter resultados de um *profiling* ao BiGGER. Adicionalmente este documento inclui detalhes em apêndice que auxiliam um contribuidor recém-integrado na tarefa de executar o algoritmo, com o intuito de familiarização com este. De notar que a falta de documentação em termos de usabilidade do Open-Chemera

e do BiGGER constituiu um obstáculo inicial para a resolução do problema associado à tese.

7.2 Trabalho futuro

Tendo em conta a quantidade de contratempos no decorrer da fase de elaboração, não foi possível finalizar a implementação da solução paralelizada de forma a garantir ganhos de desempenho face à versão sequencial do BiGGER. No entanto os problemas da solução atual encontram-se identificados, podendo ser resolvidos no período posterior à entrega e defesa da dissertação.

7.2.1 Correção dos índices no *batch*

A inserção das coordenadas dos segmentos com base no seu índice está correta quando não é preciso dividir o *batch* em partições. No entanto quando é necessário partições, os índices calculados para cada um dos segmentos não correspondem aos índices da estrutura original. Deste modo, o valor para a distância dos segmentos aos átomos pode ser incorreto. Comparando com os restantes problemas a indicar adiante, este é menos urgente devido a ser um problema que afeta a validação dos resultados obtidos e não o desempenho da solução. Para corrigir o problema é necessário determinar os índices no *batch* original e à medida que as partições são iteradas copiar os índices originais necessários para o *batch* particionado.

7.2.2 Interoperabilidade entre os gestores da *heap*

A execução da solução atual pode devolver falhas de segmentação quando acaba as primeiras rotações. A origem destas falhas deve-se à existência de dois gestores diferentes da *heap* para a execução da *Marrow* e do BiGGER, não existindo interoperabilidade entre estes. Deste modo a execução de um cenário de *docking* na sua integridade é impossível, pelo que a obtenção dos resultados presentes no capítulo anterior deu-se por simulação de uma execução completa.

A solução para garantir que existe interoperabilidade entre os dois gestores consiste em garantir que a memória necessária é alocada apenas em uma componente da solução, podendo ser o BiGGER ou o *Marrow_protein_docking*. Considerando que se opta por este último, é necessário passar a iteração das rotações para o *Marrow_Protein_Docking*. No entanto as funções e procedimentos alheios à etapa de digitalização para grelhas tridimensionais que são invocados pelo ciclo devem ser passados em conjunto com este, o que dificulta a tarefa. No entanto é mais fácil aplicar o *debugging* ao código do BiGGER usando esta alternativa. Uma segunda possibilidade consiste em criar uma unidade Pascal no BiGGER destinada à alocação da memória, de forma a não ser necessário introduzir modificações ao código original do programa.

7.2.3 Correção do número de átomos usados

A solução atual não consegue ter ganhos de *speedup* em relação à versão sequencial devido ao facto de estar a considerar um número de segmentos e de átomos em excesso comparando com a versão sequencial. O número de segmentos a passar encontra-se corrigido, no entanto continua por corrigir o número de átomos a considerar por cada segmento. Os detalhes relacionados com a função *isInnerPoint* referidos em 4.3.2 foram descobertos pouco tempo antes da entrega do presente documento, no momento em que foi possível estudar os ganhos de desempenho da solução final. Até este ponto foi inferido que o BiGGER sequencial iterava as coordenadas de todos os átomos para avaliar se um segmento é interno ou externo.

A solução para resolver este problema consiste em garantir que para cada segmento apenas são usados os átomos da sua fatia para determinar a distância em alternativa a usar todos os átomos. Essa informação consta na estrutura *FHashGrid*, que deve ser passada como argumento na função *getZLine* em conjunção com a matriz de coordenadas de átomos (*FPoints*) que atualmente é passada. Será necessário incluir na *Marrow* mecanismos de filtragem de elementos de uma coleção por localização geográfica, sendo que até ao momento existe o esqueleto *filter* que não tem este comportamento implementado.

7.3 Considerações finais

O trabalho em termos gerais revelou ser desafiador, sendo necessário adquirir conhecimentos de áreas científicas diferentes das de Informática. De notar que a dissertação foi atribuída a um mês e meio da entrega e discussão do documento de preparação. Deste modo, na fase de preparação apenas foi possível estudar o conceito de *docking* de proteínas, as suas etapas e os casos de estudo relacionados com a dissertação, ficando de parte detalhes como a confirmação dos problemas do BiGGER através de uma atividade de *profiling*. Como tal existiram os seguintes contratempos:

Profilling e análise de resultados ao BiGGER A falta de documentação relacionada com a usabilidade da componente BiGGER do Open-Chemera fez com que esta atividade fosse demasiado extensa. Outro fator a considerar foi a dificuldade encontrada em compilar e executar o BiGGER em modo *profiling*. A maior parte das alternativas existentes para efetuar o *profiling* ao BiGGER resultaram em falhas de segmentação, apenas funcionado com o Valgrind/Callgrind.

Instabilidade da *Marrow* e erros de execução OpenCL A dinamização das estruturas usadas na solução paralelizada trouxe instabilidade à *Marrow* em ambiente Linux. A execução dos testes unitários assim como dos testes usando as interações terminavam abruptamente com erros de execução específicos ao OpenCL. Deste modo não foi possível a obtenção de resultados usando interações realistas e respetiva avaliação, levando à necessidade de entrar no período de rescrita. A resolução destes erros

ocorreu pouco tempo antes da entrega da versão rescrita.

Compilação e execução do BiGGER no *cluster* Após o período de entrega da versão não-rescrita foi concedido o acesso ao *cluster* computacional do DI-FCT/UNL para efetuar testes ao BiGGER. Tentar executar ambas solução paralelizada e versão sequencial no *cluster* com o sistema Condor [56] demorou demasiado tempo[1]. Até ao momento não foi possível executar ambas as versões do BiGGER usando o Condor. A execução da versão sequencial do BiGGER no *cluster* sem o Condor ocorre sem problemas. No entanto a execução da versão paralelizada termina com erros de OpenCL por parte da *Marrow*, sendo um contratempo adicional.

Apesar destes contra-tempos terem ocorrido a maior parte dos objetivos indicados foram alcançados. Por sua vez destacam-se como pontos positivos o desenvolvimento das aptidões de programação nas linguagens C++ e Pascal.

[1] A tarefa demorou os dois meses anteriores à data de entrega da versão rescrita

Bibliografia

[1] J. Adler. "Chemotaxis in bacteria". Em: *Science* 153.3737 (1966), pp. 708–716.

[2] F. Alexandre, R. Marques e H. Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". Em: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 880–885.

[3] R. Almeida, S. Dell'Acqua, L. Krippahl, J. Moura e S. Pauleta. "Predicting Protein-Protein Interactions Using BiGGER: Case Studies". Em: 21 (ago. de 2016), p. 1037.

[4] R. M. Almeida, S. Dell'Acqua, L. Krippahl, J. J. Moura e S. R. Pauleta. "Predicting protein-protein interactions using bigger: Case studies". Em: *Molecules* 21.8 (2016), p. 1037.

[5] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov e P. E. Bourne. "The Protein Data Bank". Em: *Nucleic Acids Research* 28.1 (jan. de 2000), pp. 235–242. ISSN: 0305-1048. DOI: 10.1093/nar/28.1.235. eprint: http://oup.prod.sis.lan/nar/article-pdf/28/1/235/9895144/280235.pdf. URL: https://doi.org/10.1093/nar/28.1.235.

[6] R. Chen, L. Li e Z. Weng. "ZDOCK: an initial-stage protein-docking algorithm". Em: *Proteins: Structure, Function, and Bioinformatics* 52.1 (2003), pp. 80–87.

[7] *Containers*. URL: http://www.cplusplus.com/reference/stl/.

[8] *Dynamic array*. URL: http://wiki.freepascal.org/Dynamic_array.

[9] D. F. Elliott e K. R. Rao. *Fast transforms algorithms, analyses, applications*. Elsevier, 1983.

[10] M. W. Gonzalez e M. G. Kann. "Protein interactions and disease". Em: *PLoS computational biology* 8.12 (2012), e1002819.

[11] P Gouet, N Chinardet, M Welch, V Guillet, S Cabantous, C Birck, L Mourey e J.-P. Samama. "Further insights into the mechanism of function of the response regulator CheY from crystallographic studies of the CheY–CheA124–257 complex". Em: *Acta Crystallographica Section D: Biological Crystallography* 57.1 (2001), pp. 44–51.

[12] *GPGPU.org*. URL: http://gpgpu.org/about.

[13] M. Harris. *Six Ways to SAXPY*. URL: https://devblogs.nvidia.com/six-ways-saxpy/.

[14] C. I. "Distributed programming using AGAPIA". Em: *International Journal of Advanced Computer Science and Applications* 5 (jan. de 2014). DOI: 10.14569/IJACSA.2014.050304.

[15] H. Inbal, M. Buyong, W. Haim e N. Ruth. *Principles of docking: An overview of search algorithms and a guide to scoring functions*. Vol. 47. 4, pp. 409–443. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.10115.

[16] T. K. G. Inc. *Errors*. URL: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/errors.html.

[17] S Kannan e R Ganji. "Porting AutoDock to CUDA [J]. Evolutionary Computation (CEC)". Em: *2010 IEEE Congress on*, pp. 1–8.

[18] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo e I. A. Vakser. "Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques." Em: *Proceedings of the National Academy of Sciences* 89.6 (1992), 2195–2199. DOI: 10.1073/pnas.89.6.2195.

[19] T. G. Kolda e B. W. Bader. "Tensor decompositions and applications". Em: *SIAM review* 51.3 (2009), pp. 455–500.

[20] L. Krippahl. *Integrating protein structural information*. 2003. URL: https://userweb.fct.unl.pt//\~a4338/pdfs/LK_PhD_HR.pdf.

[21] R. Landaverde e M. C. Herbordt. "GPU optimizations for a production molecular docking code". Em: *... IEEE conference on high performance extreme computing. IEEE Conference on High Performance Extreme Computing*. Vol. 2014. NIH Public Access. 2014.

[22] E. Lindholm, J. Nickolls, S. Oberman e J. Montrym. "NVIDIA Tesla: A unified graphics and computing architecture". Em: *IEEE micro* 28.2 (2008).

[23] G. Marcou, E. Engler e A. Varnek. *How to use C code in Free Pascal projects*. 2009. URL: http://ftp://ftp.freepascal.org/fpc/docs-pdf/CinFreePascal.pdf.

[24] R. Marques. *Algorithmic Skeleton Framework for the Orchestration of GPU Computations*. 2012.

[25] R. Marques, H. Paulino, F. Alexandre e P. D. Medeiros. "Algorithmic skeleton framework for the orchestration of GPU computations". Em: *European Conference on Parallel Processing*. Springer. 2013, pp. 874–885.

[26] *Marrow Benchmarks*. URL: https://bitbucket.org/marrow-project/marrow-benchmarks/src/master/.

[27] M. D. McCool. "Structured parallel programming with deterministic patterns". Em: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association. 2010, pp. 5–5.

[28] G. M. Morris. *AutoDock*. URL: http://autodock.scripps.edu/.

[29] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew e A. J. Olson. "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function". Em: *Journal of computational chemistry* 19.14 (1998), pp. 1639–1662.

[30] J. Nickolls e W. J. Dally. "The GPU computing era". Em: *IEEE micro* 30.2 (2010).

[31] V. Novais. *Equipa liderada por português descobre proteína do cérebro que protege de Alzheimer*. URL: https://observador.pt/2018/06/29/ha-uma-proteina-do-cerebro-que-pode-proteger-contra-a-doenca-de-alzheimer/.

[32] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[33] NVIDIA. *CUDA C PROGRAMMING GUIDE*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[34] NVIDIA. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone.

[35] NVIDIA. *GPU Aplications Catalog*. URL: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/.

[36] NVIDIA. *NVIDIA Tesla V100 GPU architecture*. URL: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[37] NVIDIA. *OpenCL*. URL: https://developer.nvidia.com/opencl.

[38] NVIDIA. *cuFFT*. 2018. URL: https://developer.nvidia.com/cufft.

[39] M. Ohue, T. Shimoda, S. Suzuki, Y. Matsuzaki, T. Ishida e Y. Akiyama. "MEGADOCK 4.0: an ultra–high-performance protein–protein docking software for heterogeneous supercomputers". Em: *Bioinformatics* 30.22 (2014), pp. 3281–3283. URL: http://dx.doi.org/10.1093/bioinformatics/btu532.

[40] O. Osakwe e S. A. Rizvi. *Social aspects of drug discovery, development and commercialization*. Academic Press, 2016.

[41] P. N. Palma, L. Krippahl, J. E. Wampler e J. Moura. "BIGGER: A new (soft) docking algorithm for predicting protein interactions". Em: 39 (jun. de 2000), pp. 372–84.

[42] *placeholder type specifiers (since C++11)*. URL: https://en.cppreference.com/w/cpp/language/auto.

[43] *Profiling*. URL: http://wiki.lazarus.freepascal.org/Profiling.

[44] K. Rogers e R. J. Kadner. *Bacteria*. URL: https://www.britannica.com/science/bacteria/Capsules-and-slime-layers.

[45] H. Schreuder, C. Tardif, S. Trump-Kallmeyer, A. Soffientini, E. Sarubbi, A. Akeson, T. Bowlin, S. Yanofsky e R. W. Barrett. "A new cytokine-receptor binding mode revealed by the crystal structure of the IL-1 receptor with an antagonist". Em: *Nature* 386.6621 (1997), p. 194.

[46] Scigenis. *Docking (molecular)*. URL: https://en.wikipedia.org/wiki/Docking_(molecular).

[47] *SFINAE*. URL: https://en.cppreference.com/w/cpp/language/sfinae.

[48] T. Shimoda, S. Suzuki, M. Ohue, T. Ishida e Y. Akiyama. "Protein-protein docking on hardware accelerators: comparison of GPU and MIC architectures". Em: *BMC systems biology*. Vol. 9. 1. BioMed Central. 2015, S6.

[49] G. R. Smith e M. J. Sternberg. "Prediction of protein–protein interactions by docking methods". Em: *Current opinion in structural biology* 12.1 (2002), pp. 28–35.

[50] F. Soldado, F. Alexandre e H. Paulino. "Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments". Em: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787.

[51] *std::copy, std::copy_if*. URL: https://en.cppreference.com/w/cpp/algorithm/copy.

[52] *std::size_t*. URL: https://en.cppreference.com/w/cpp/types/size_t.

[53] J. E. Stone, D. Gohara e G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". Em: *Computing in Science and Engineering* 12.3 (2010), pp. 66–73.

[54] B. Sukhwani e M. C. Herbordt. "GPU acceleration of a production molecular docking code". Em: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM. 2009, pp. 19–27.

[55] O. Trott e A. J. Olson. "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading". Em: *Journal of computational chemistry* 31.2 (2010), pp. 455–461.

[56] UW-Madison. *Computing with HTCondor*. URL: https://research.cs.wisc.edu/htcondor/.

[57] I. A. Vakser. "Protein-protein docking: From interaction to interactome". Em: *Biophysical journal* 107.8 (2014), pp. 1785–1793.

[58] T. Vreven, I. H. Moal, A. Vangone, B. G. Pierce, P. L. Kastritis, M. Torchala, R. Chaleil, B. Jiménez-García, P. A. Bates, J. Fernandez-Recio et al. "Updates to the integrated protein–protein interaction benchmarks: docking benchmark version 5 and affinity benchmark version 2". Em: *Journal of molecular biology* 427.19 (2015), pp. 3031–3041. URL: https://zlab.umassmed.edu/benchmark/.

[59] E. W. Weisstein. "Euler angles". Em: (2009). URL: http://mathworld.wolfram.com/EulerAngles.html.

[60] N. Wilt. *The CUDA handbook: a comprehensive guide to GPU programming*. Addison-Wesley, 2013.

[61] E. Zhang, R. S. Charles e A Tulinsky. “Structure of extracellular tissue factor complexed with factor VIIa inhibited with a BPTI mutant”. Em: *Journal of molecular biology* 285.5 (1999), pp. 2089–2104.

Apêndice 1

Neste apêndice inicial estão incluídos os gráficos de linhas para os testes executados na análise de desempenho da versão sequencial do BiGGER. Existem três agrupamentos sendo o primeiro para os gráficos onde foi variado no eixo das abcissas a sobreposição mínima. No segundo agrupamento constam os gráficos onde foi variado o número de rotações. Por fim o terceiro agrupamento contém os gráficos onde o tamanho (em número de átomos) das estruturas ligando e recetor foi variado.

A.1 Gráficos de profiling

Legenda:

T<*numAtomos*> *Target* (recetor) com *numAtomos* átomos .

P<*numAtomos*> *Probe* (ligando) com *numAtomos* átomos.

R<*numRotations*> O teste utilizou *numRotations* rotações .

O<*minOverlap*> Foi adotada uma sobreposição mínima de valor *minOverlap*.

A.1.1 Variação de sobreposição mínima

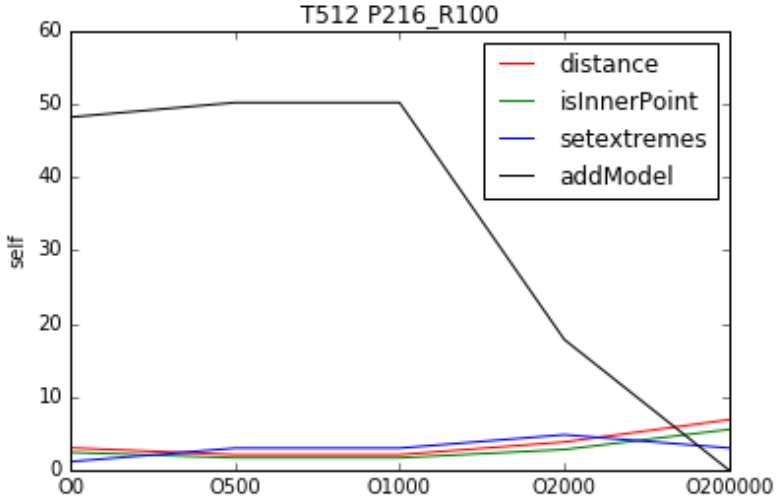

Figura A.1: T512P216R100

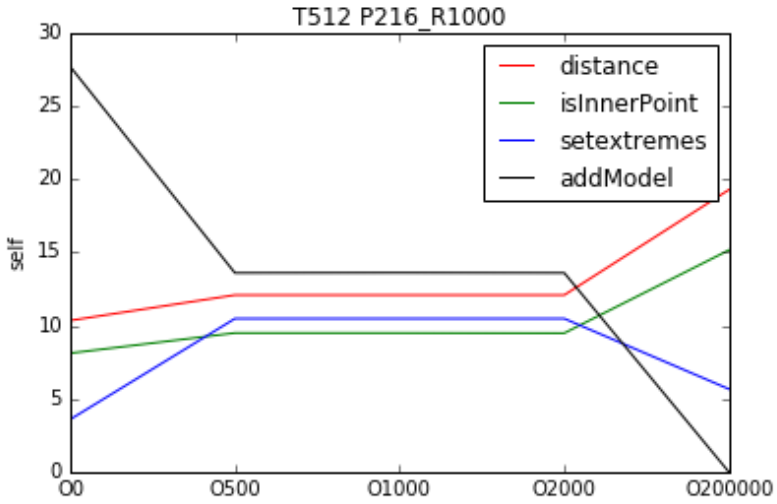

Figura A.2: T512P216R1000

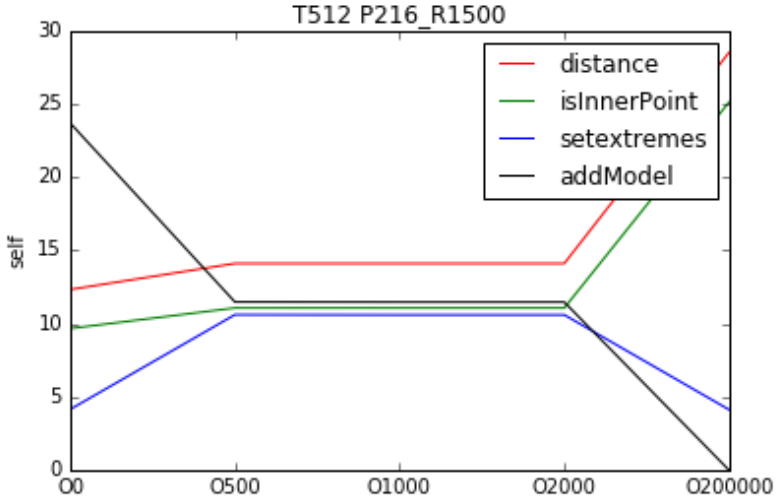

Figura A.3: T512P216R1500

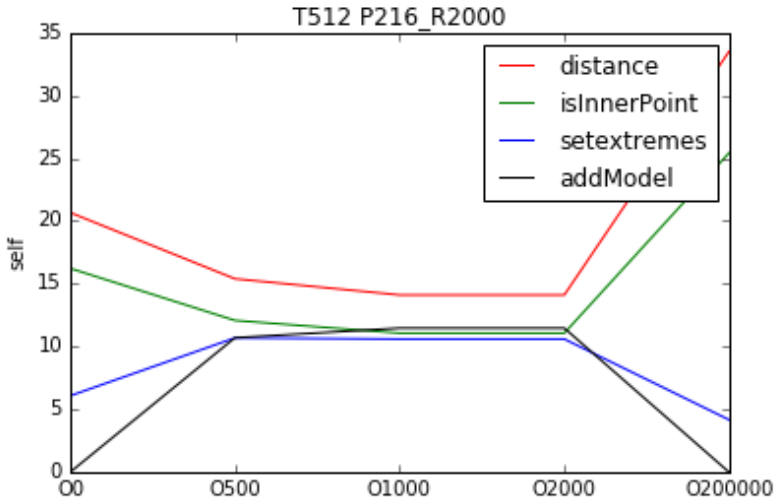

Figura A.4: T512P216R2000

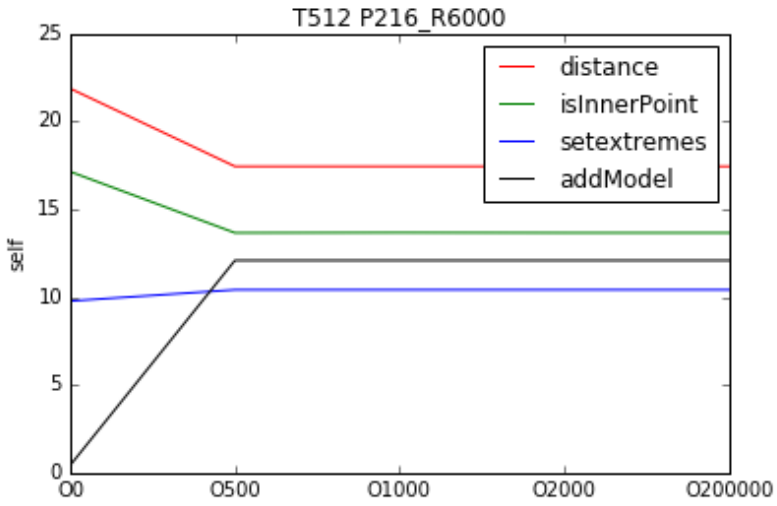

Figura A.5: T512P216R6000

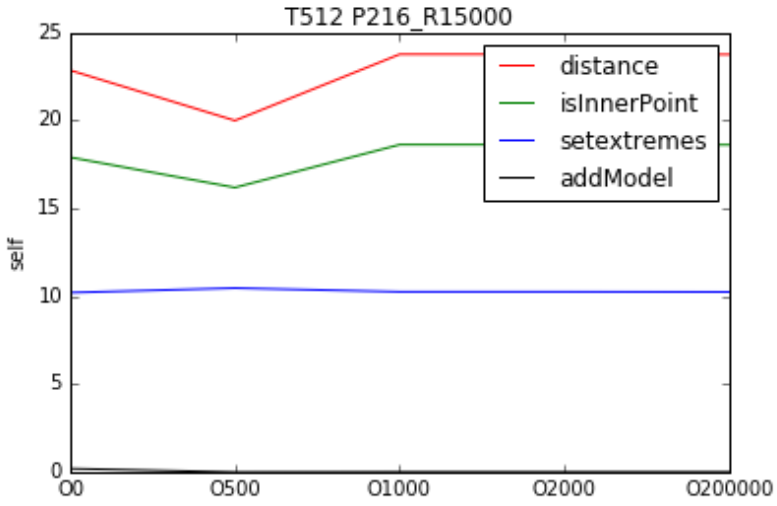

Figura A.6: T512P216R15000

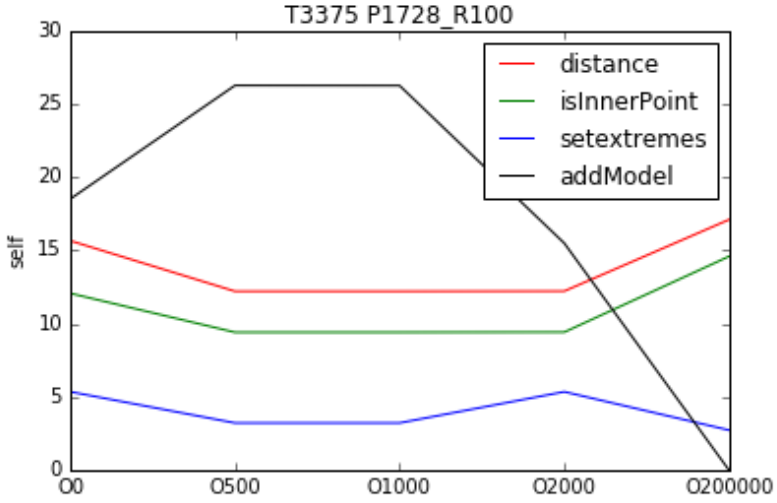

Figura A.7: T3375P1728R100

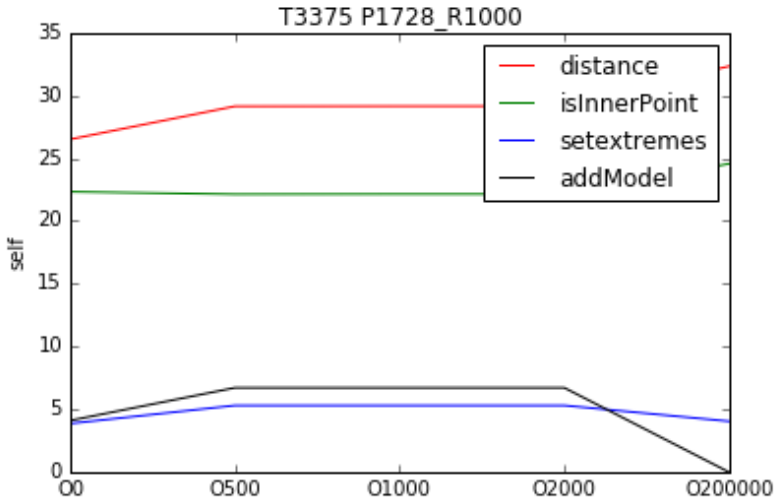

Figura A.8: T3375P1728R1000

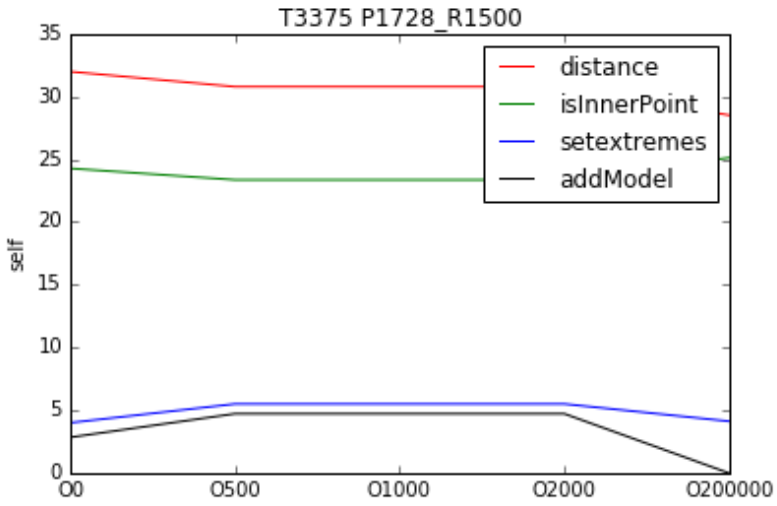

Figura A.9: T3375P1728R1500

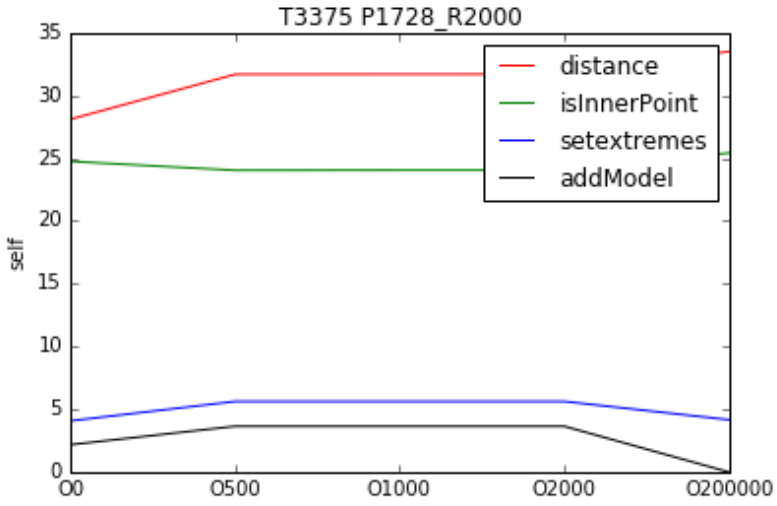

Figura A.10: T3375P1728R2000

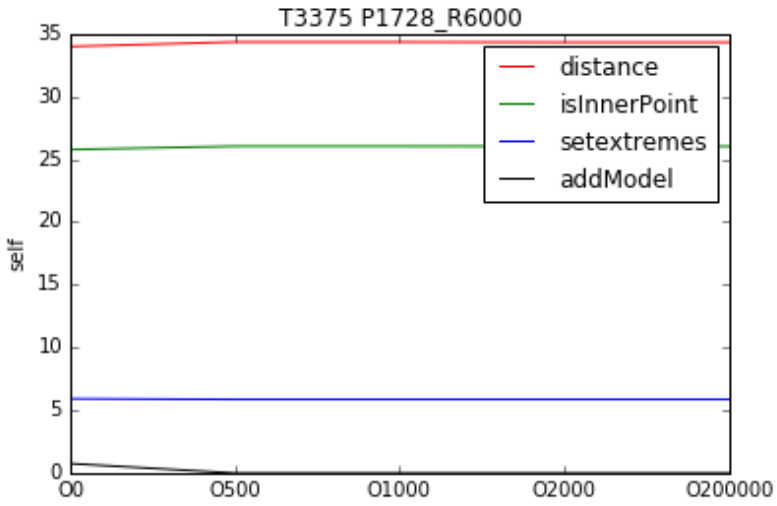

Figura A.11: T3375P1728R6000

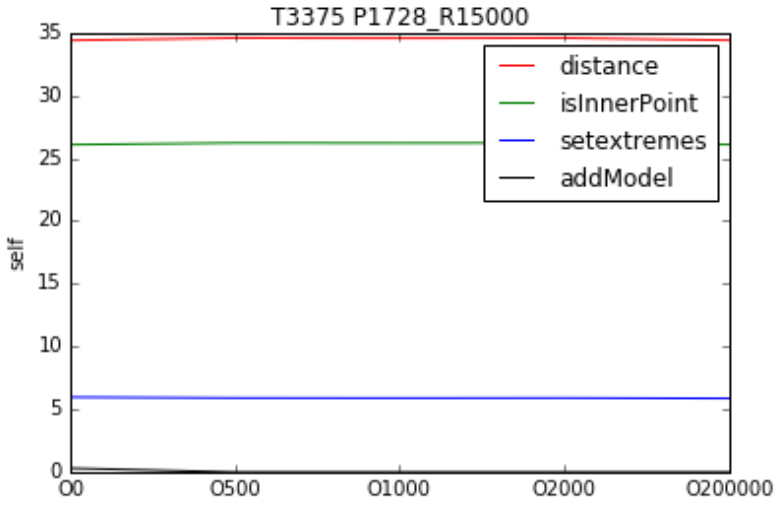

Figura A.12: T3375P1728R15000

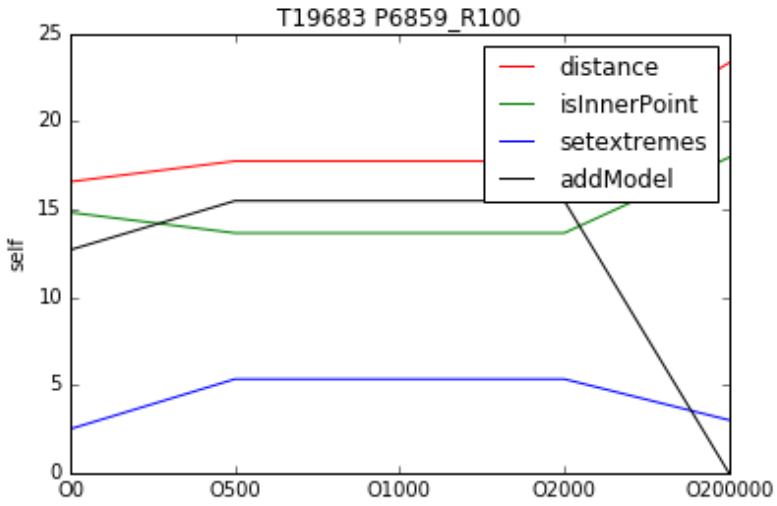

Figura A.13: T19683P6859R100

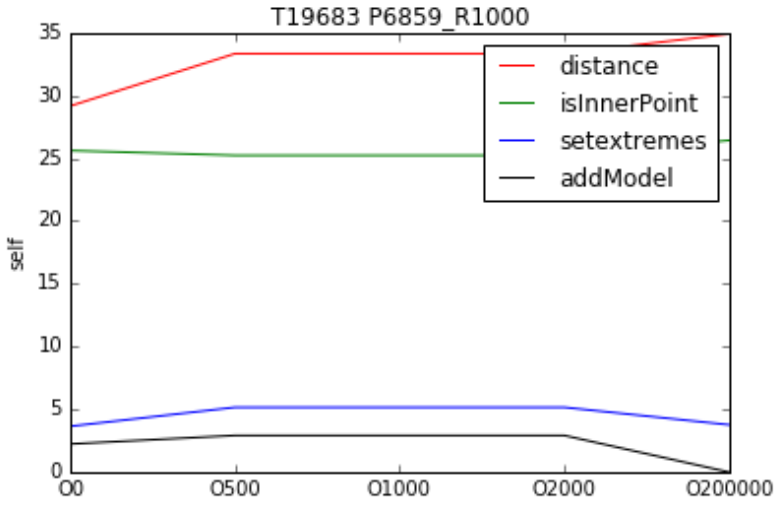

Figura A.14: T19683P6859R1000

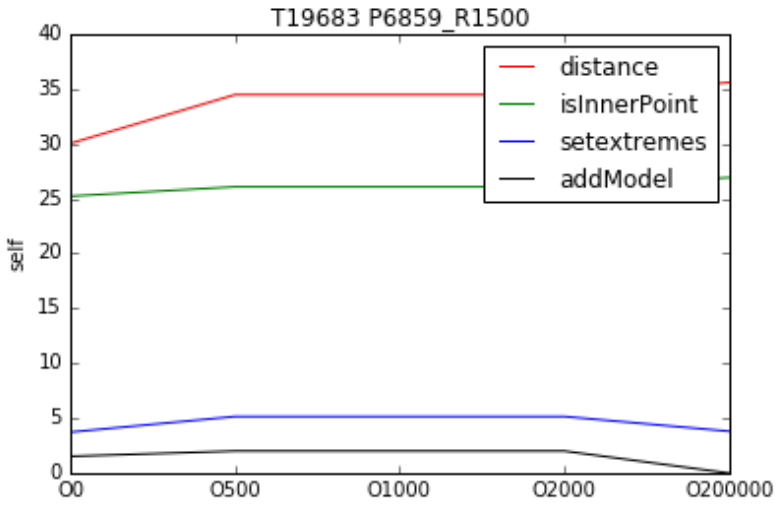

Figura A.15: T19683P6859R1500

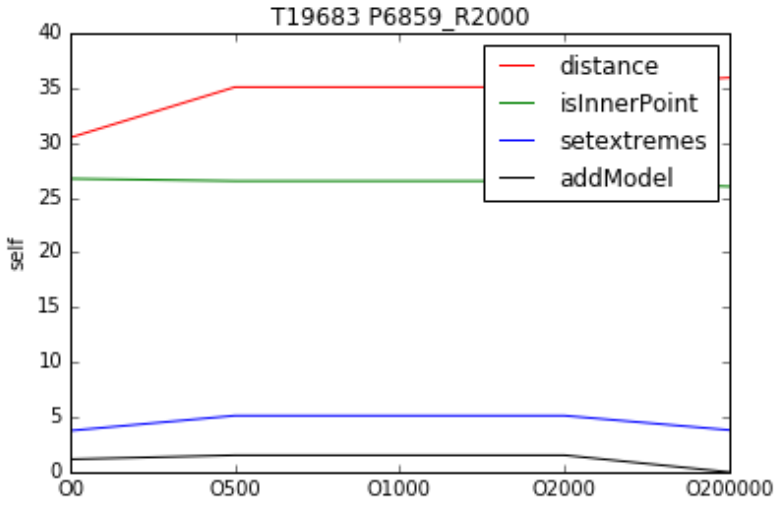

Figura A.16: T19683P6859R2000

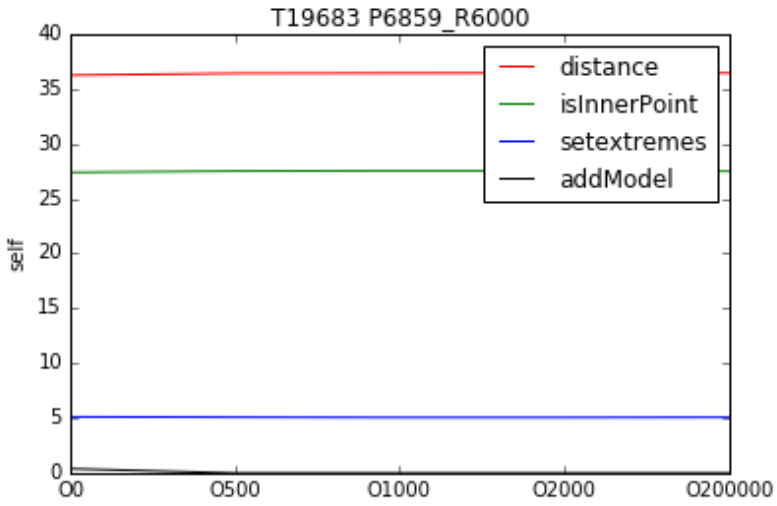

Figura A.17: T19683P6859R6000

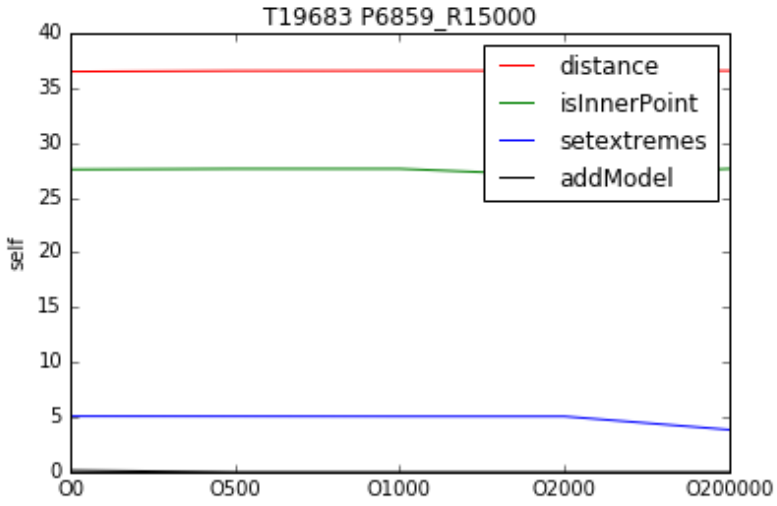

Figura A.18: T19683P6859R15000

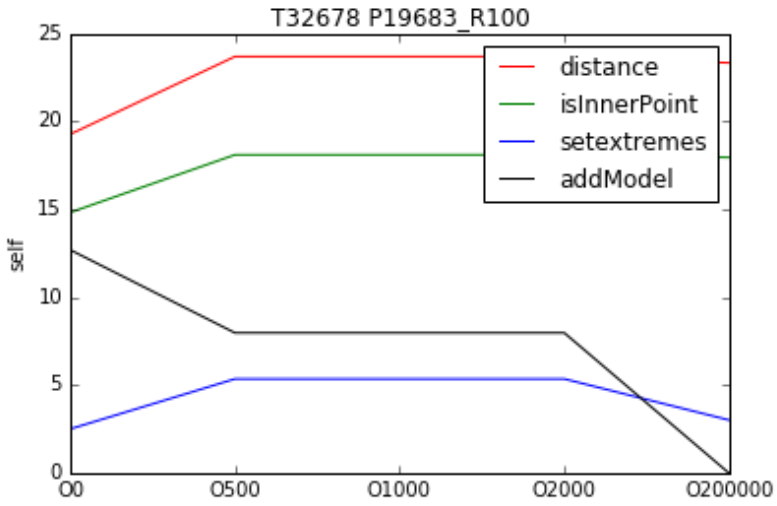

Figura A.19: T32678P19683R100

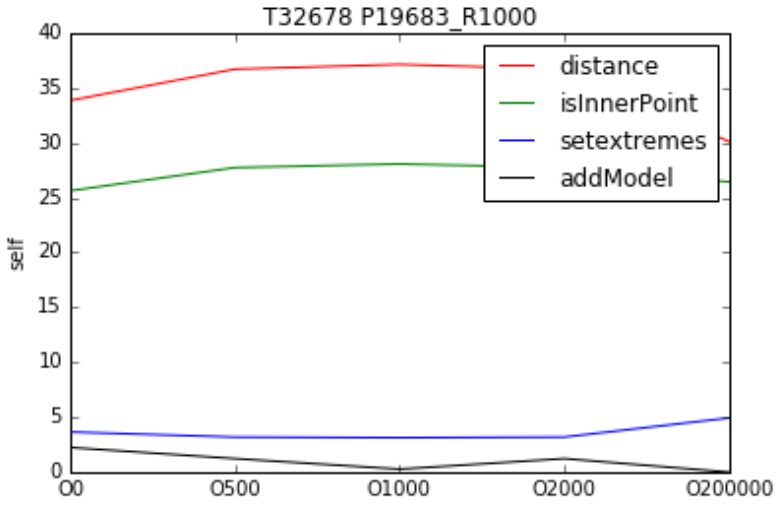

Figura A.20: T32678P19683R1000

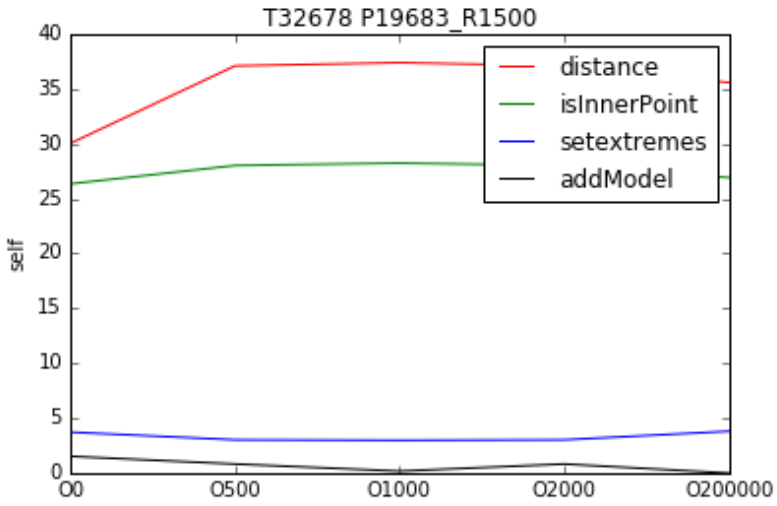

Figura A.21: T32678P19683R1500

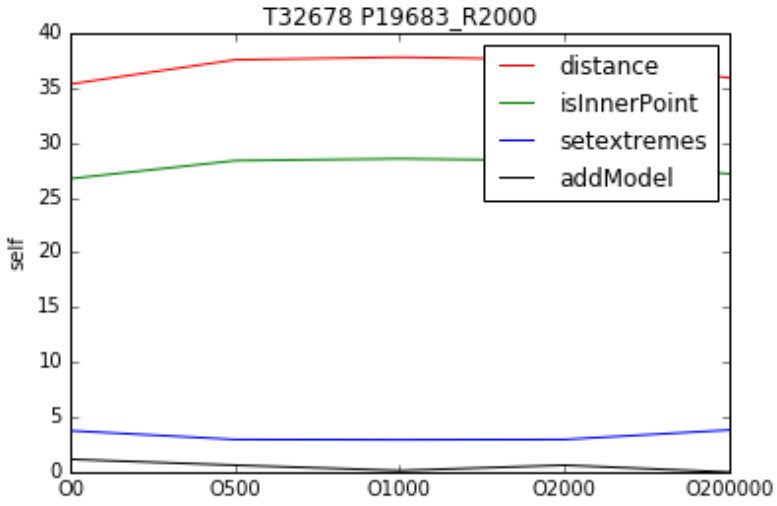

Figura A.22: T32678P19683R2000

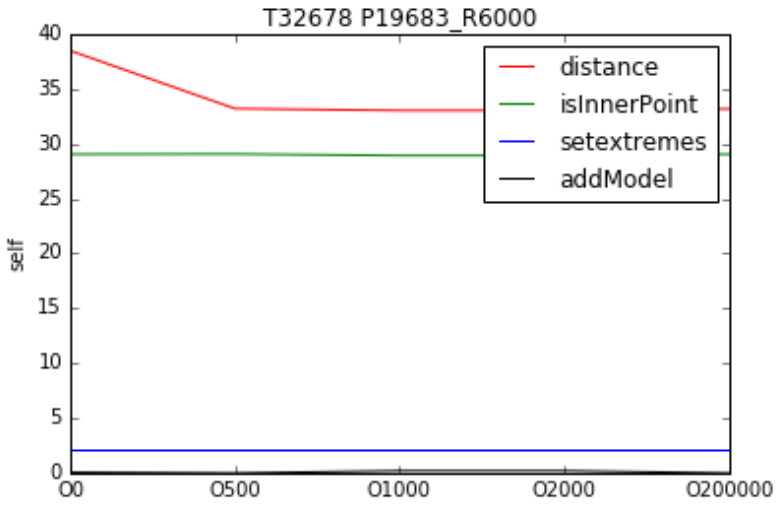

Figura A.23: T32678P19683R6000

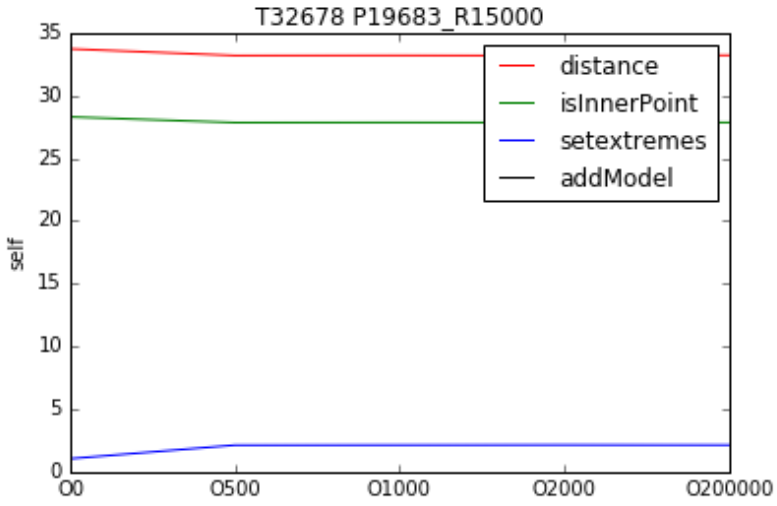

Figura A.24: T32678P19683R15000

A.1.2 Variação de rotações

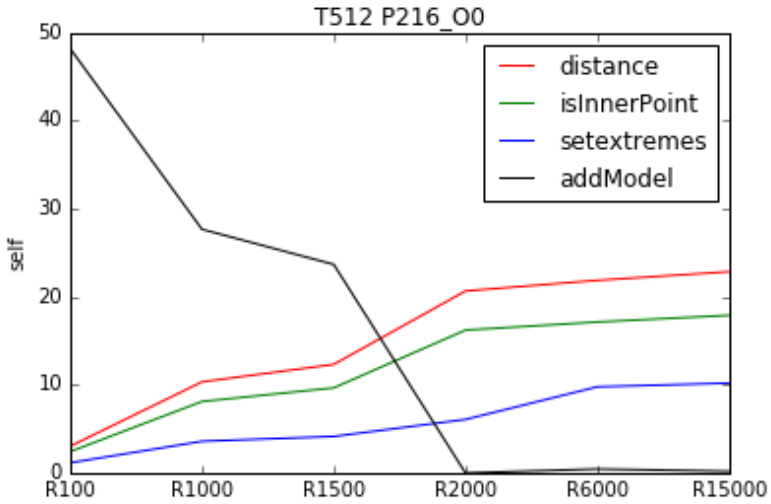

Figura A.25: T512P216O0

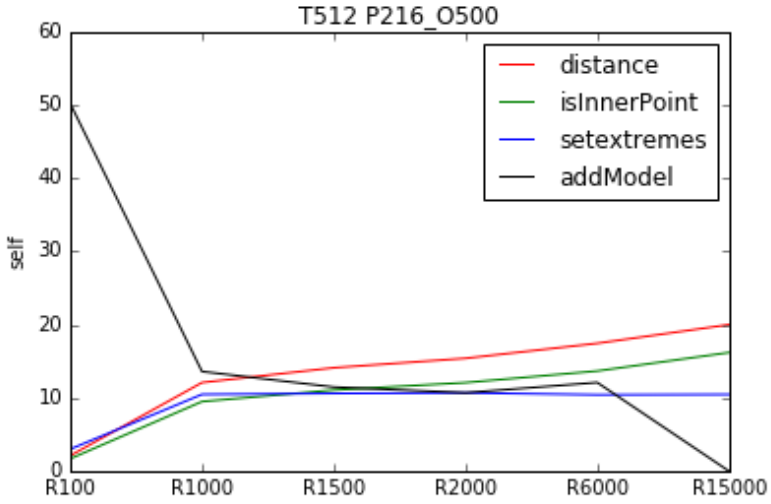

Figura A.26: T512P216O500

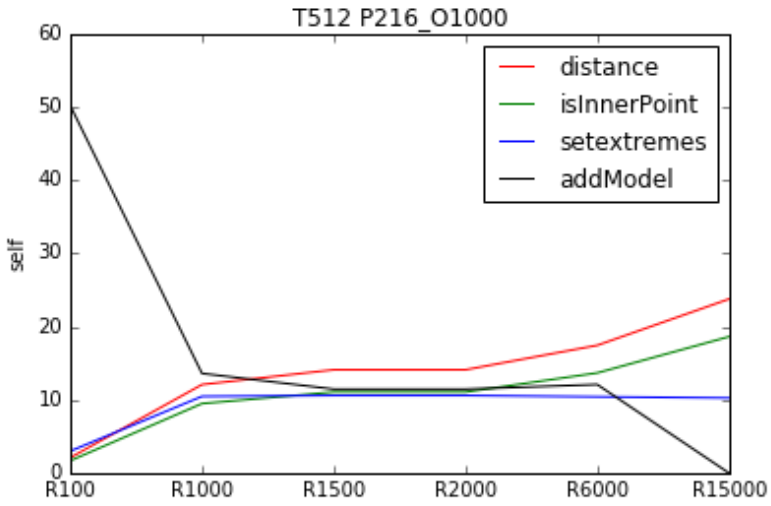

Figura A.27: T512P216O1000

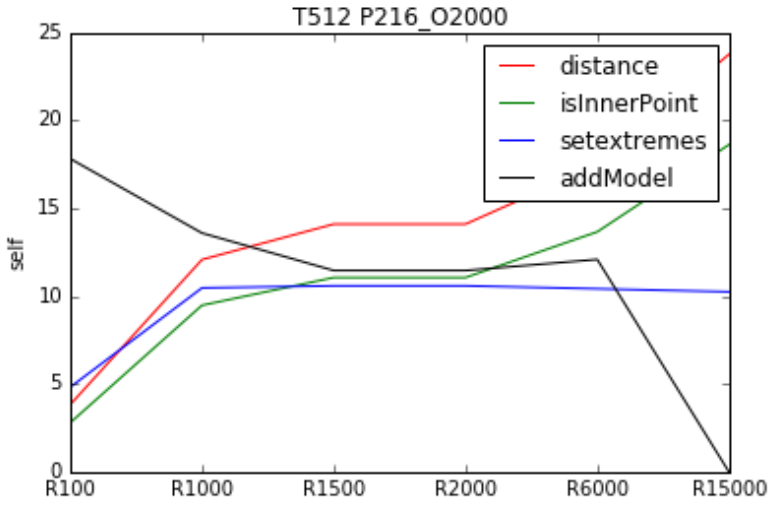

Figura A.28: T512P216O2000

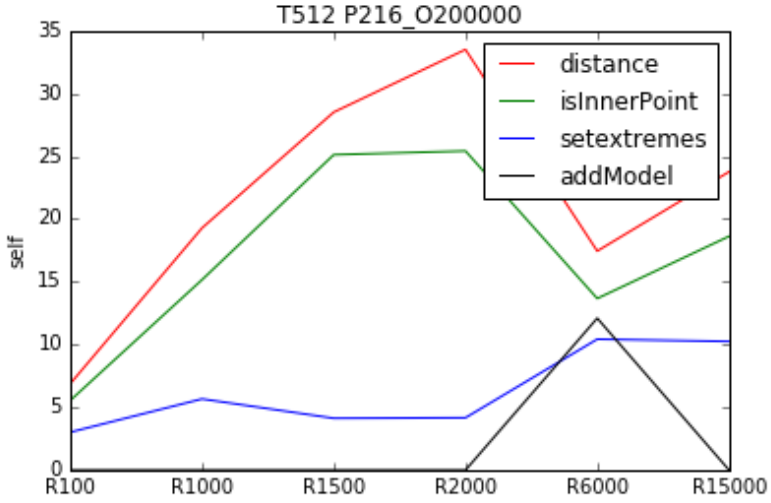

Figura A.29: T512P216O200000

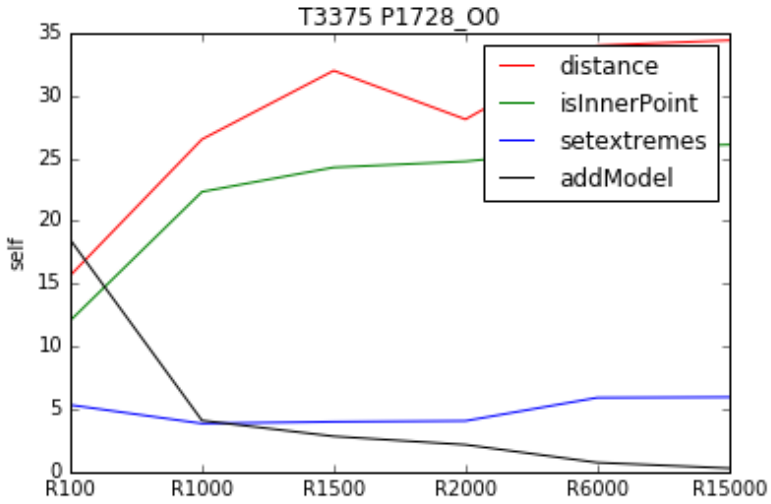

Figura A.30: T3375P1728O0

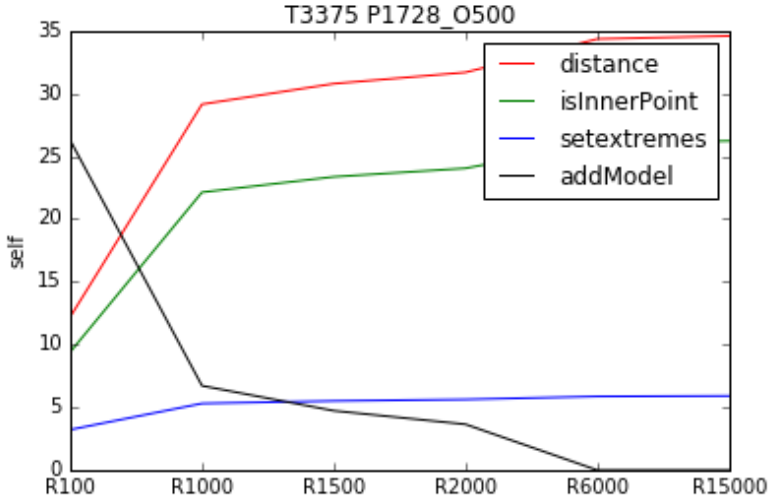

Figura A.31: T3375P1728O500

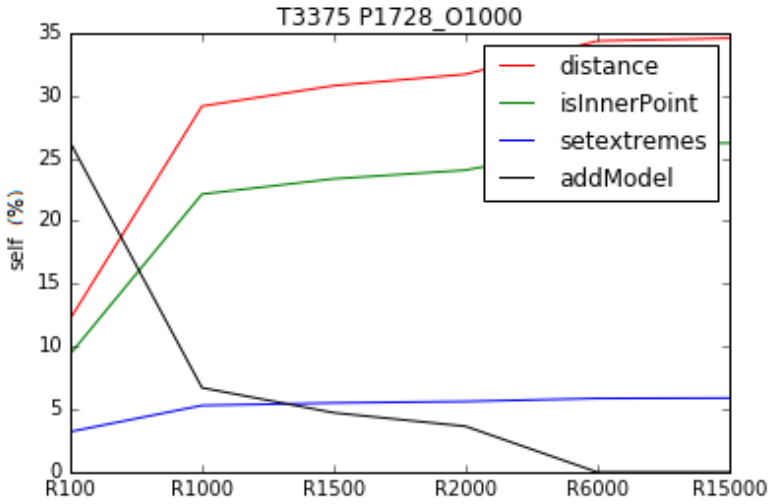

Figura A.32: T3375P1728O1000

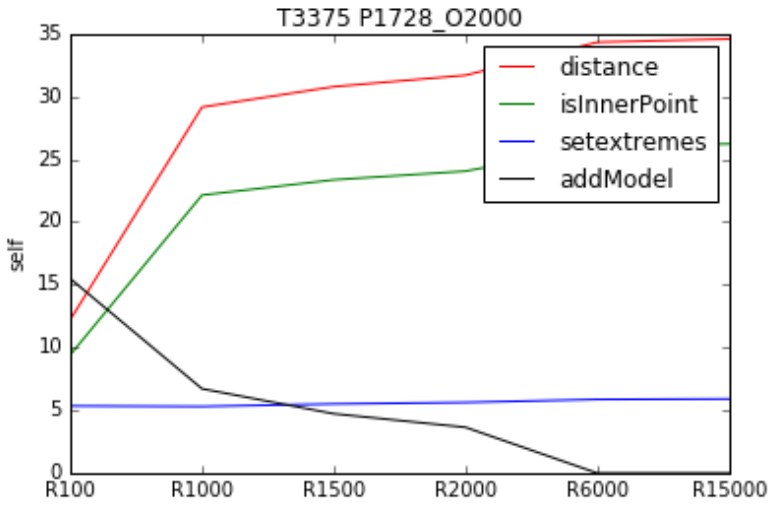

Figura A.33: T3375P1728O2000

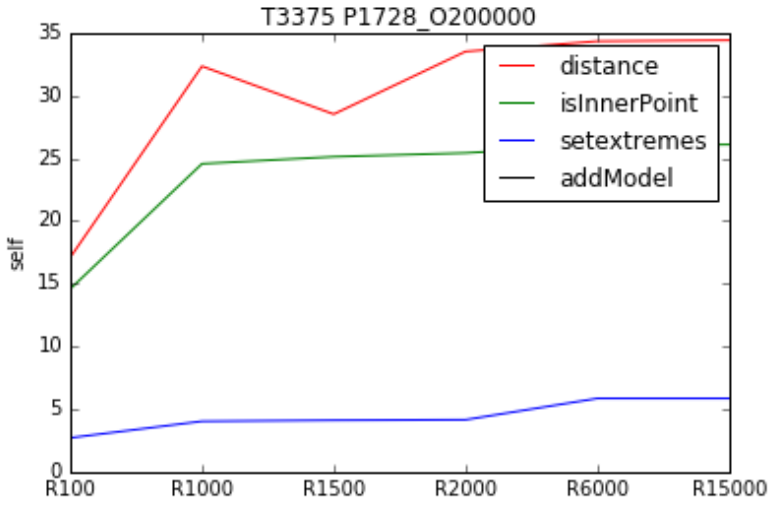

Figura A.34: T3375P1728O200000

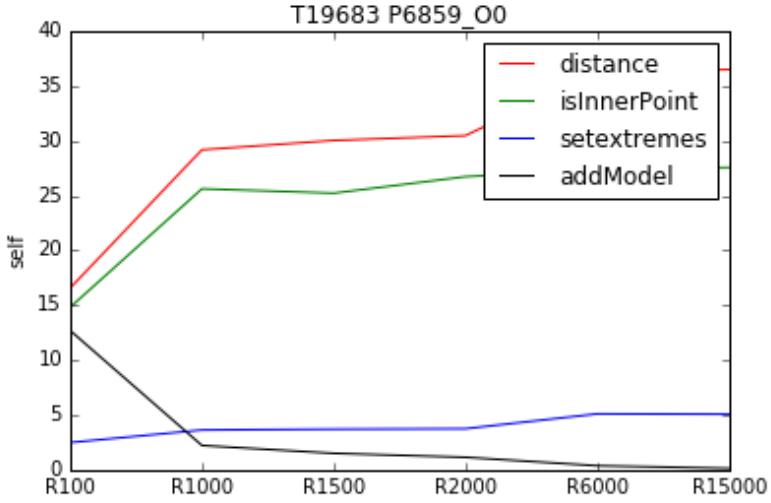

Figura A.35: T19683P6859O0

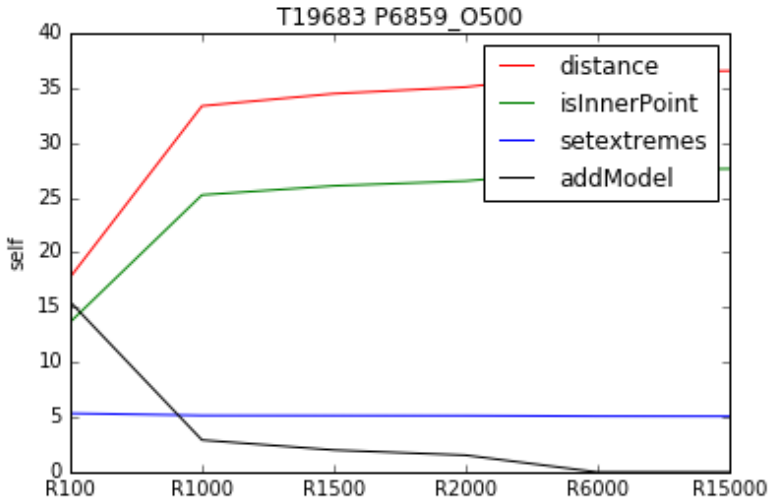

Figura A.36: T19683P6859O500

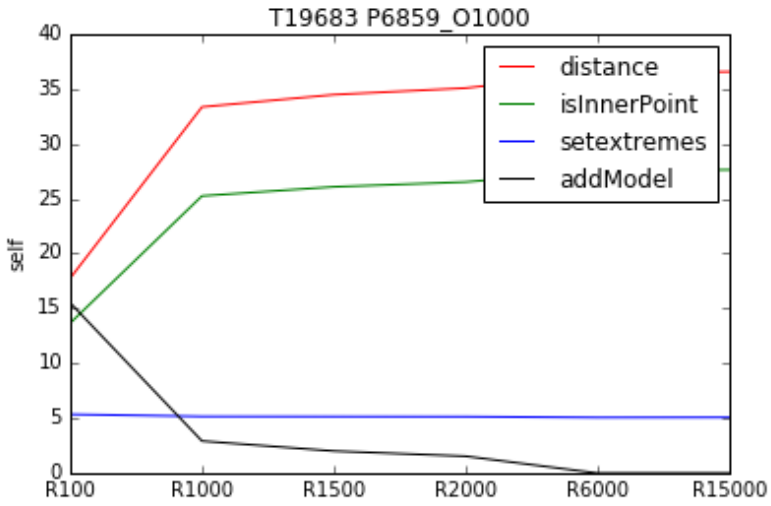

Figura A.37: T19683P6859O1000

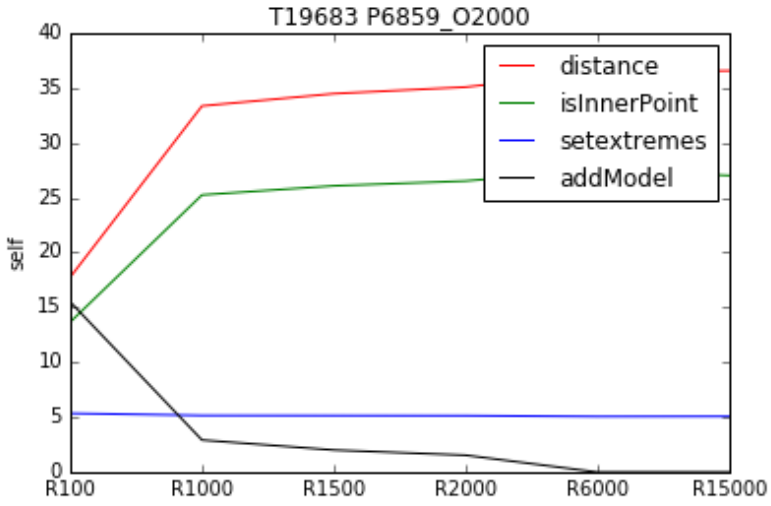

Figura A.38: T19683P6859O2000

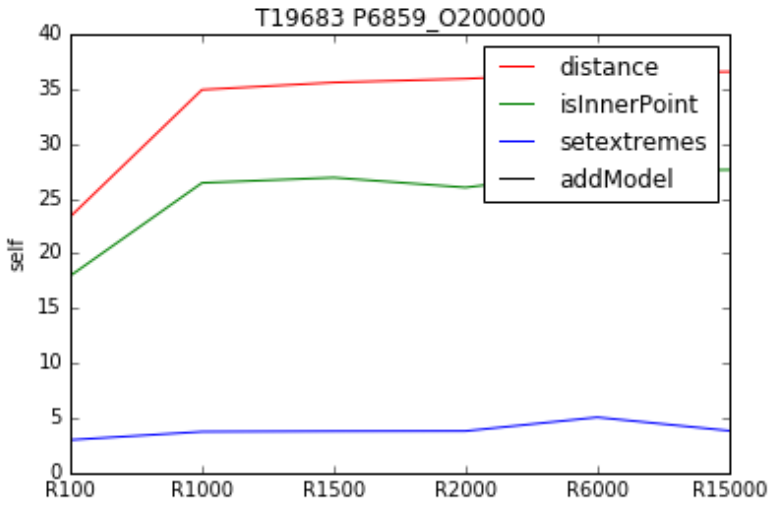

Figura A.39: T19683P6859O200000

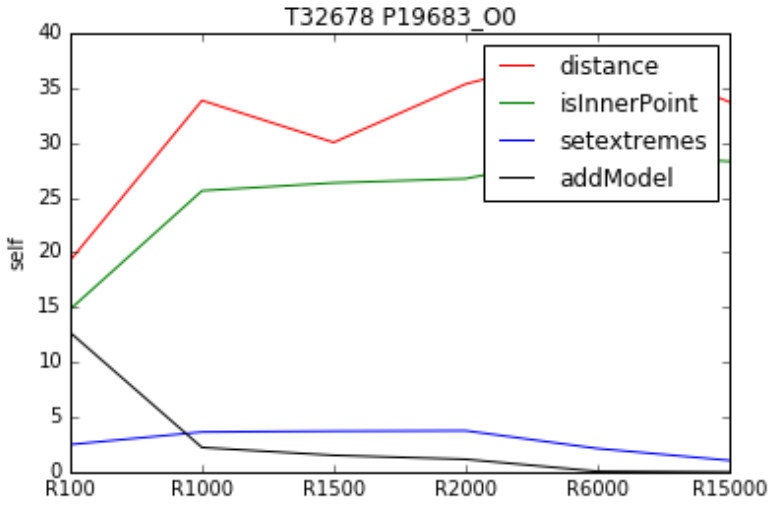

Figura A.40: T32678P19683O0

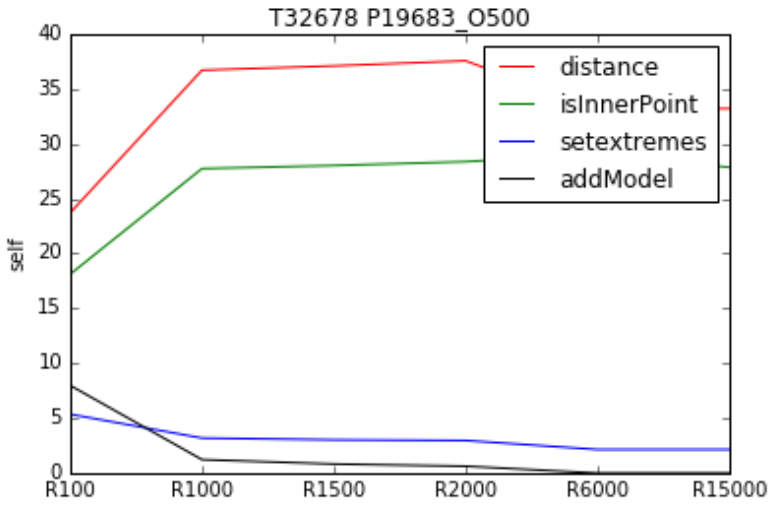

Figura A.41: T32678P19683O500

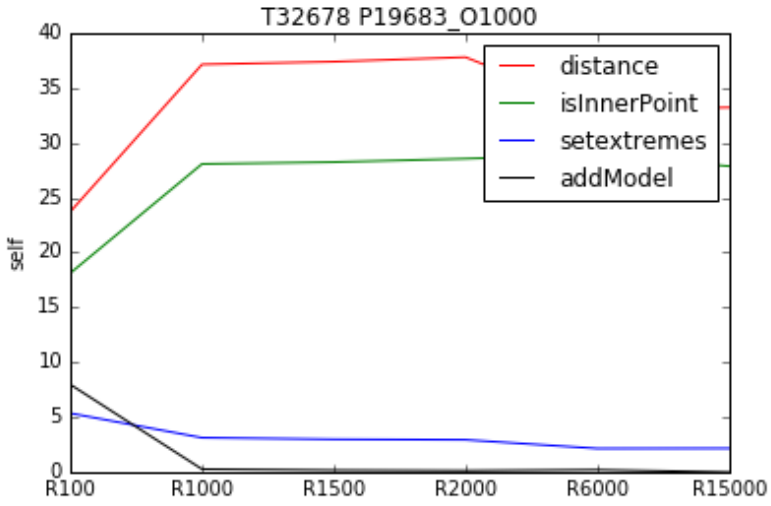

Figura A.42: T32678P19683O1000

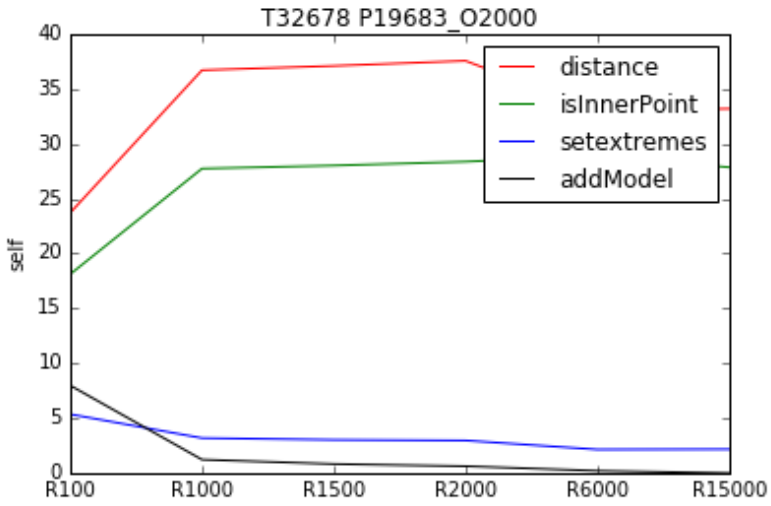

Figura A.43: T32678P19683O2000

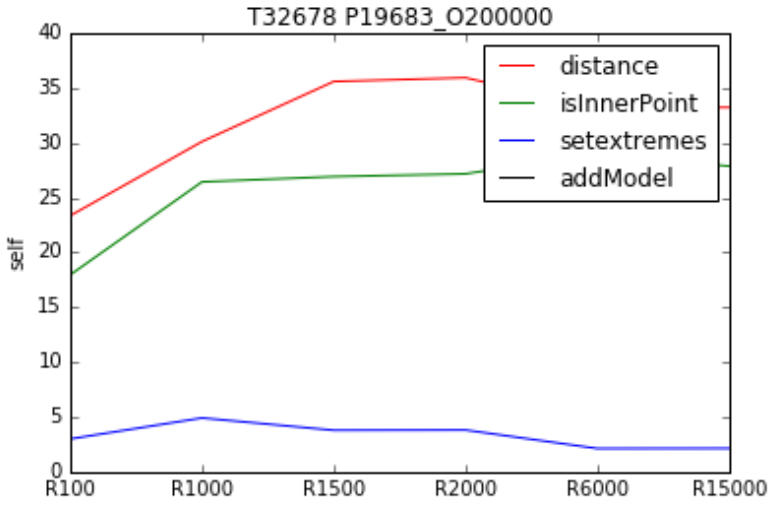

Figura A.44: T32678P19683O200000

A.1.3 Variação do número de átomos nas proteínas

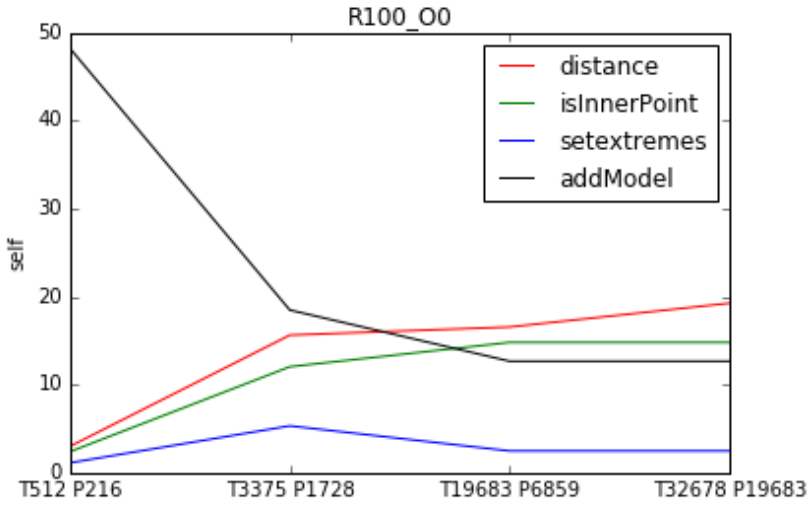

Figura A.45: R100O0

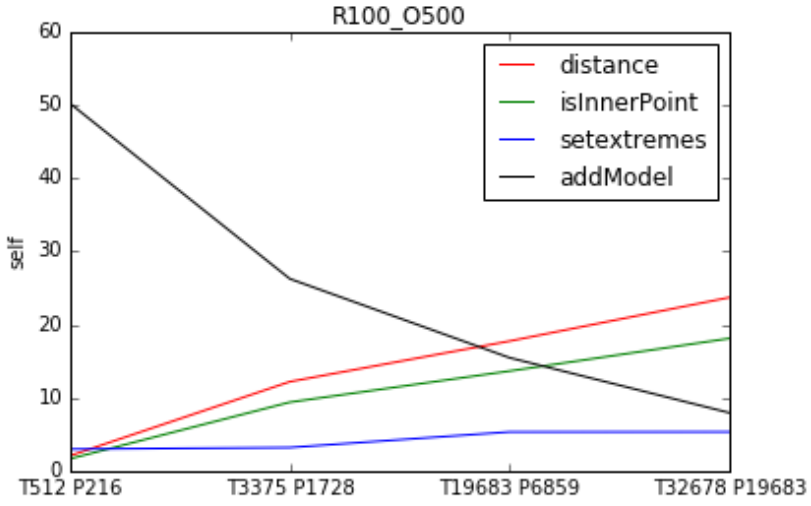

Figura A.46: R100O500

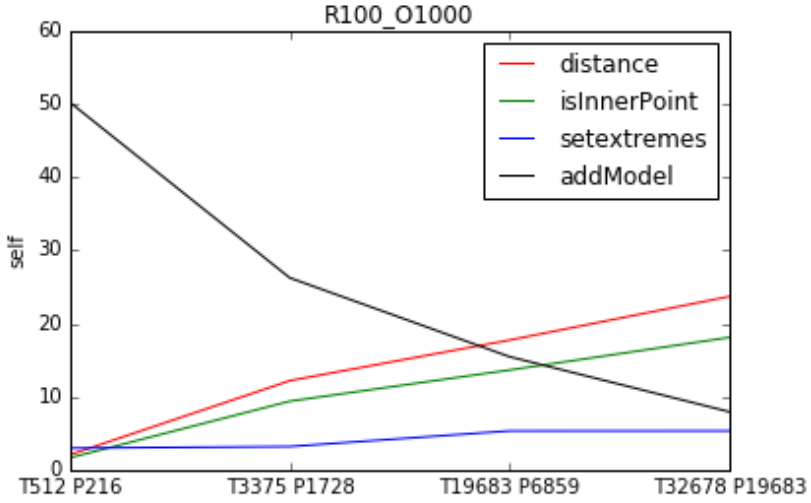

Figura A.47: R100O1000

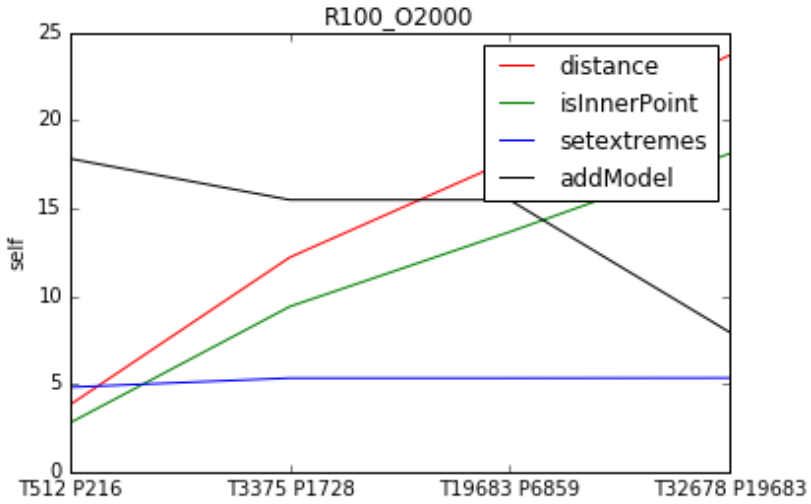

Figura A.48: R100O2000

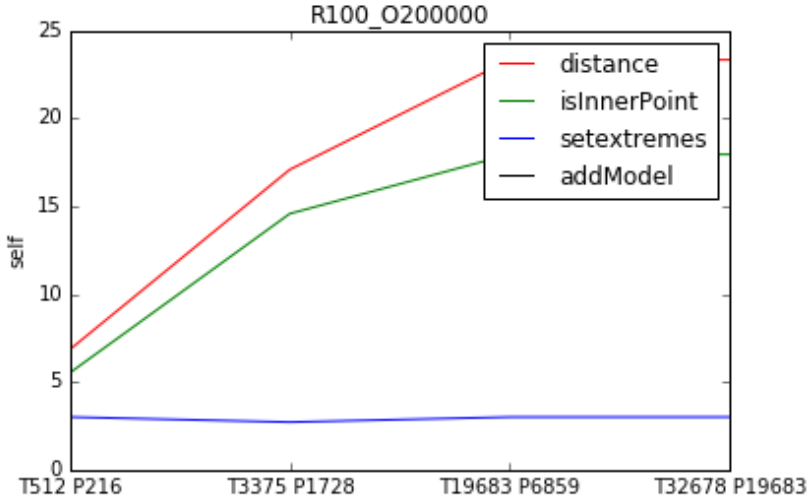

Figura A.49: R100O200000

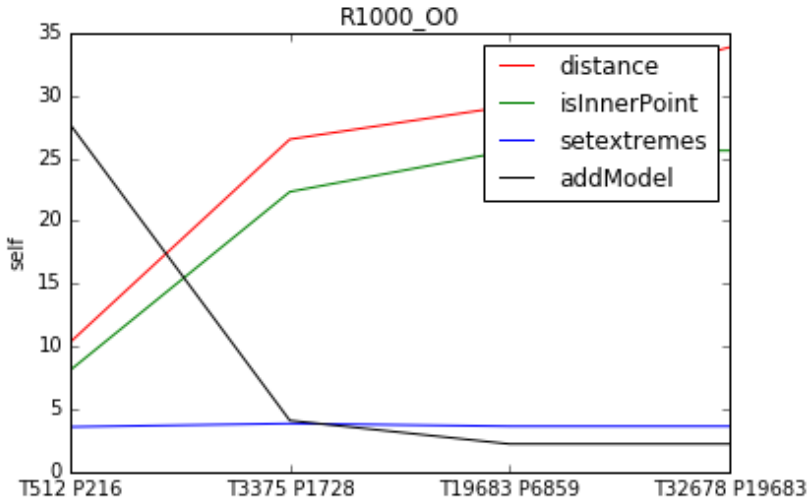

Figura A.50: R1000O0

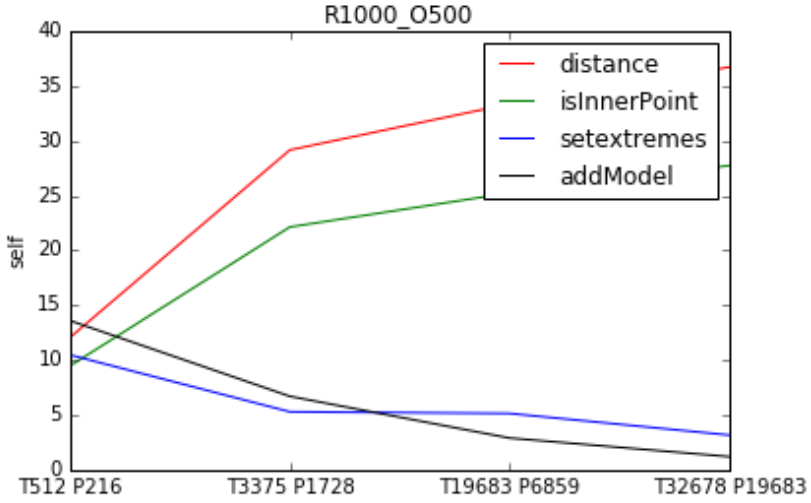

Figura A.51: R1000O500

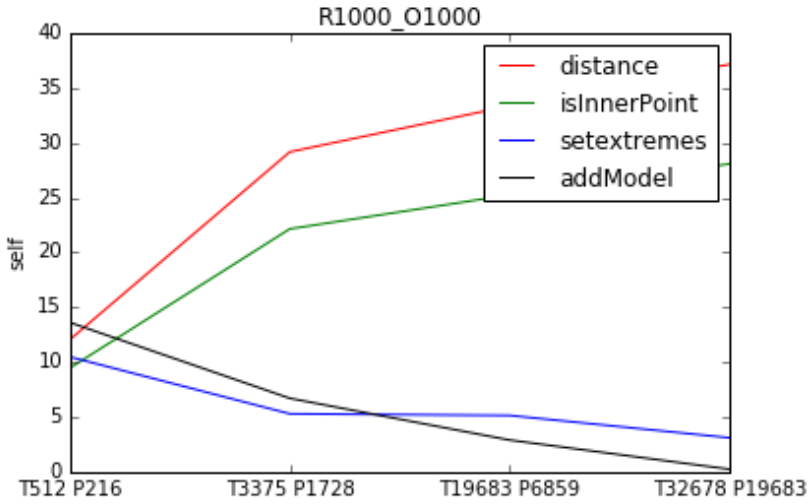

Figura A.52: R1000O1000

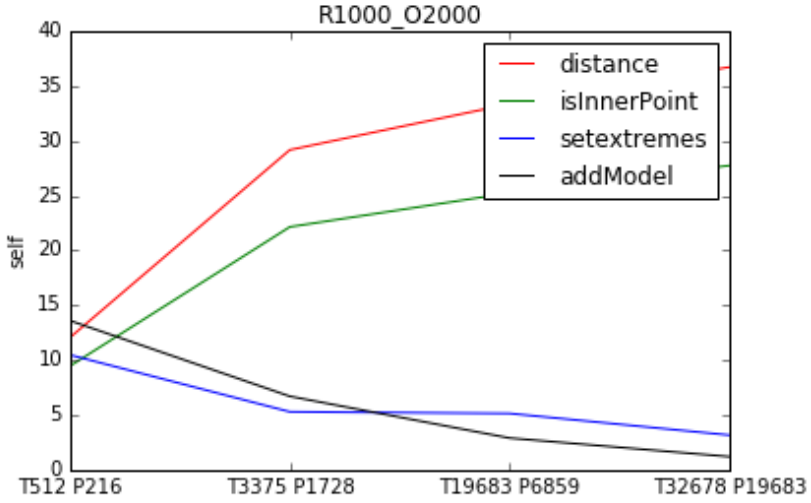

Figura A.53: R1000O2000

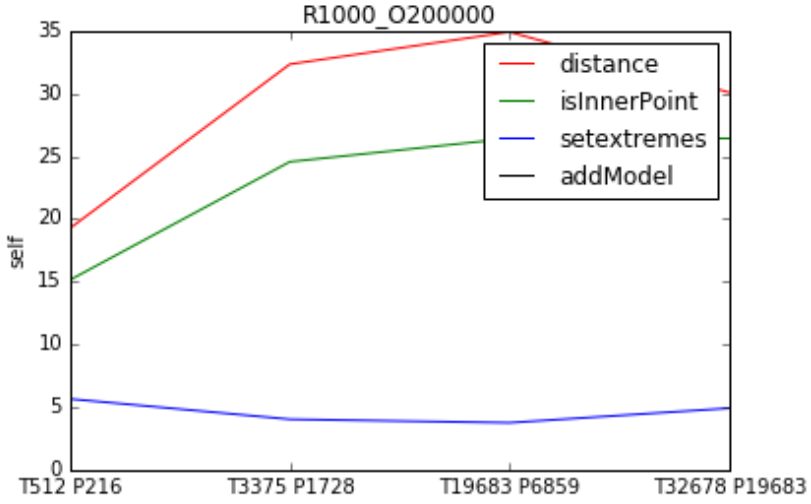

Figura A.54: R1000O200000

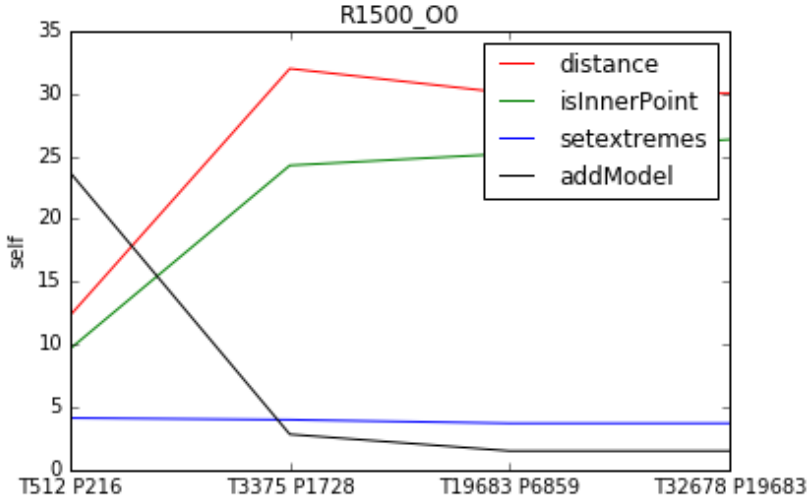

Figura A.55: R1500O0

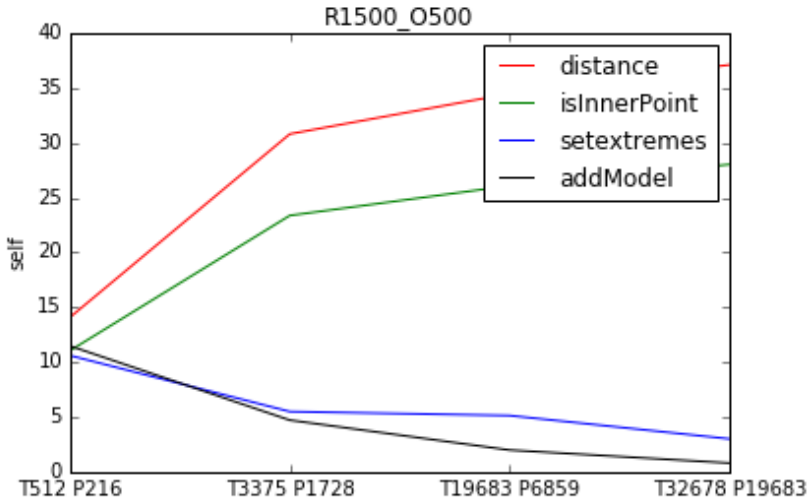

Figura A.56: R1500O500

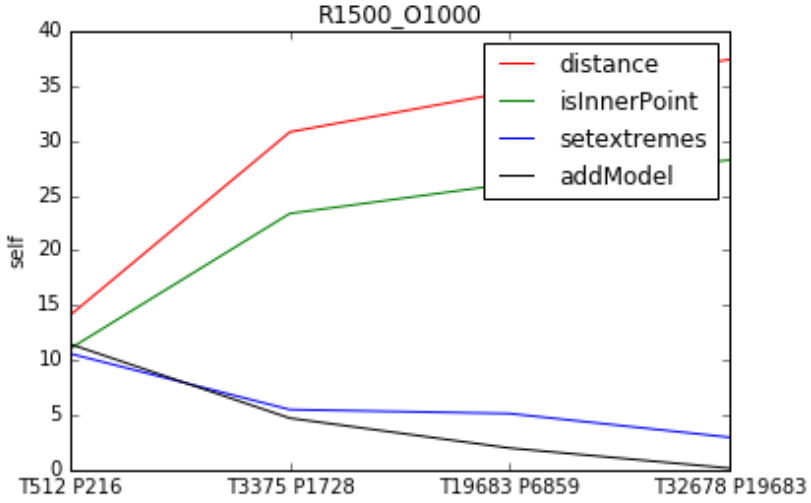

Figura A.57: R1500O1000

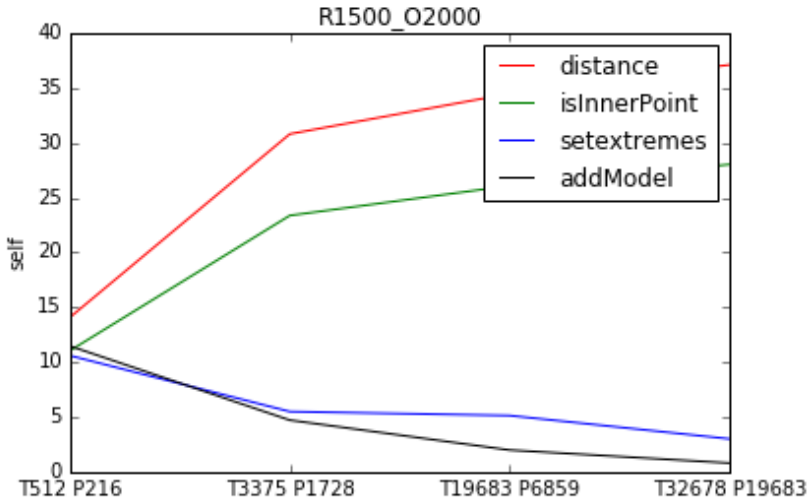

Figura A.58: R1500O2000

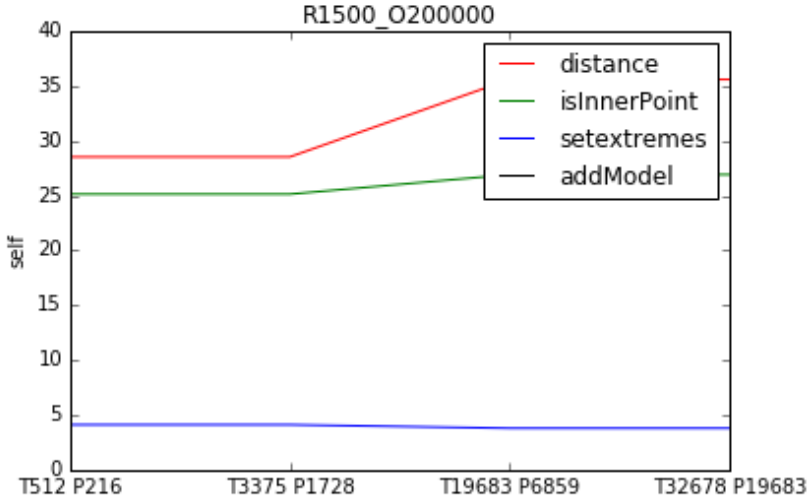

Figura A.59: R1500O200000

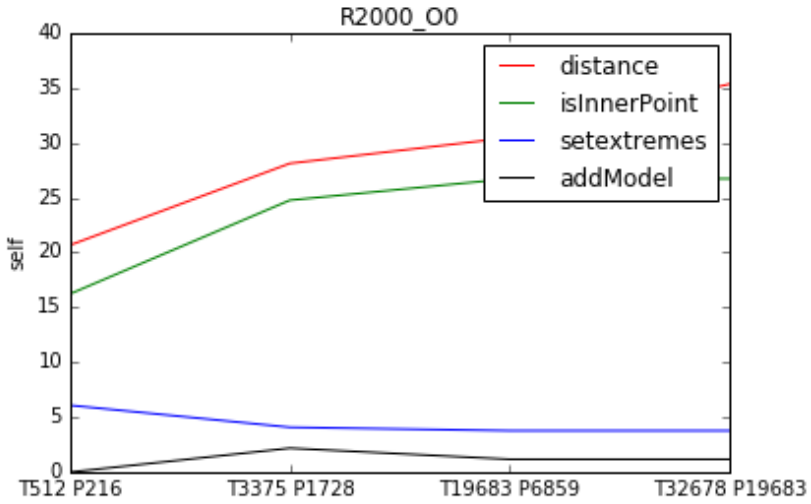

Figura A.60: R2000O0

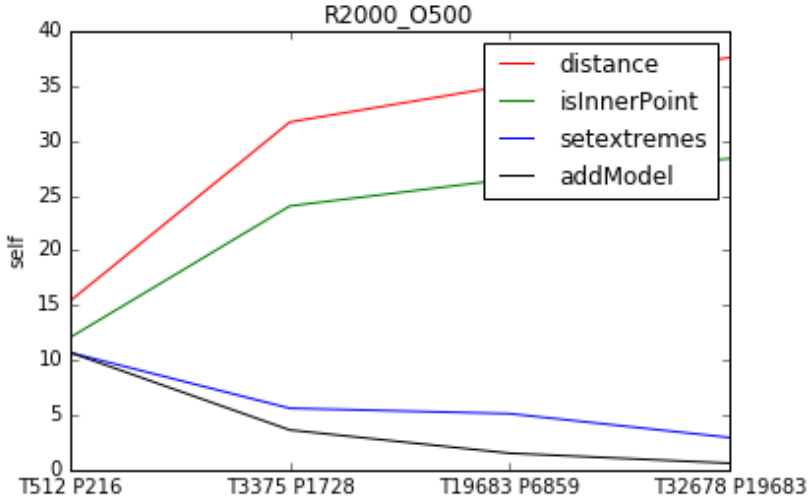

Figura A.61: R2000O500

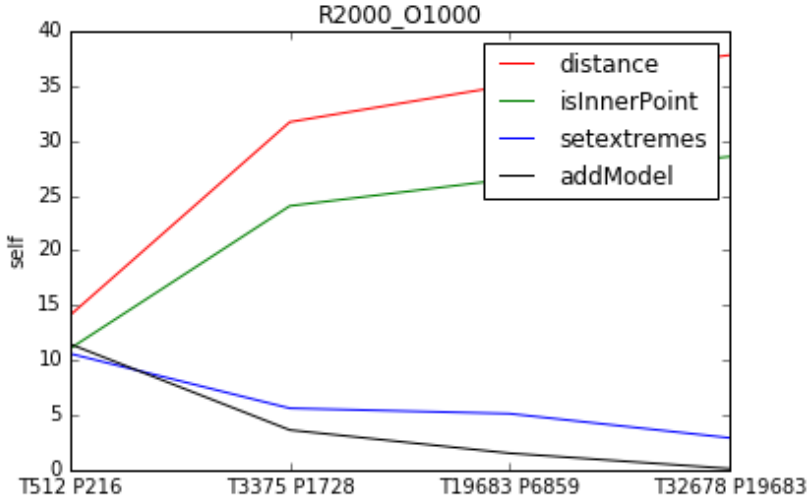

Figura A.62: R2000O1000

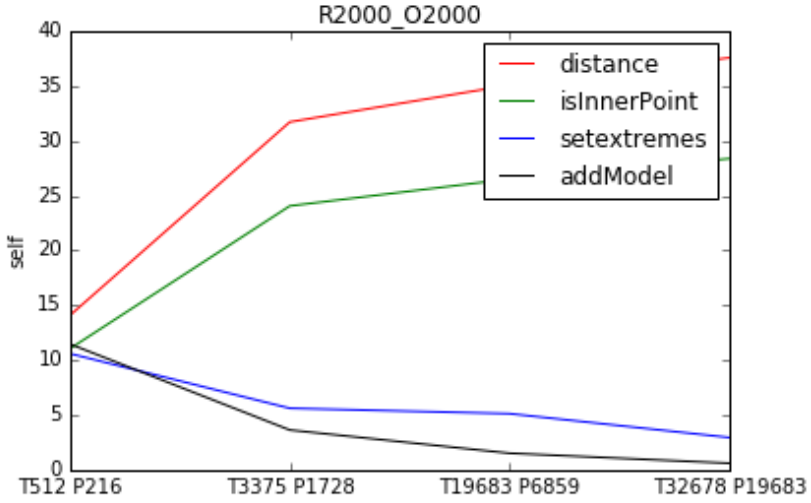

Figura A.63: R2000O2000

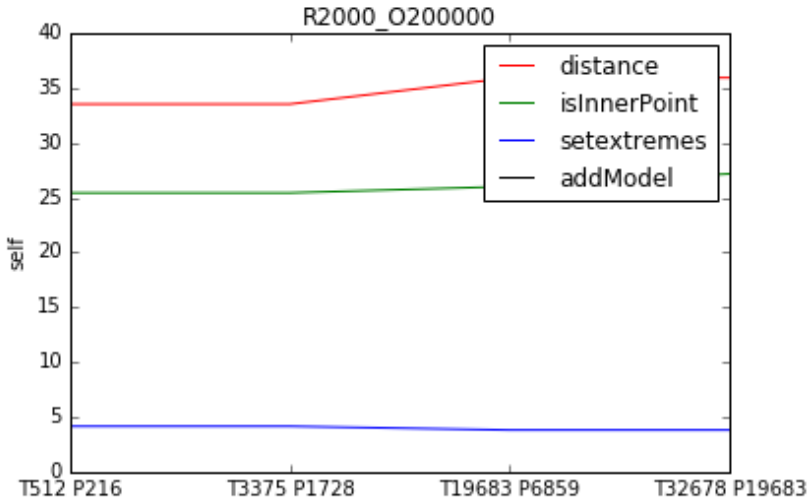

Figura A.64: R2000O200000

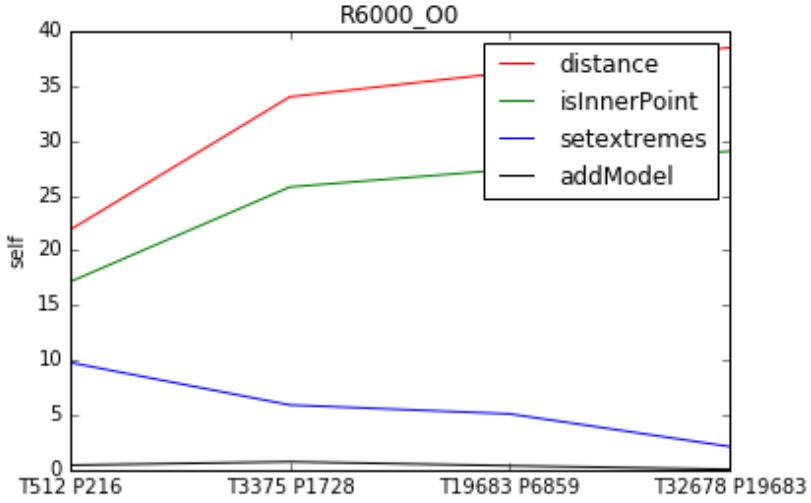

Figura A.65: R6000O0

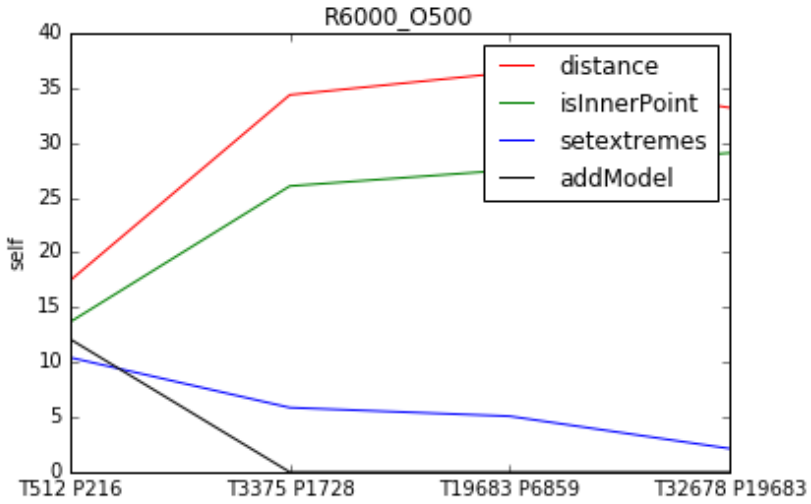

Figura A.66: R6000O500

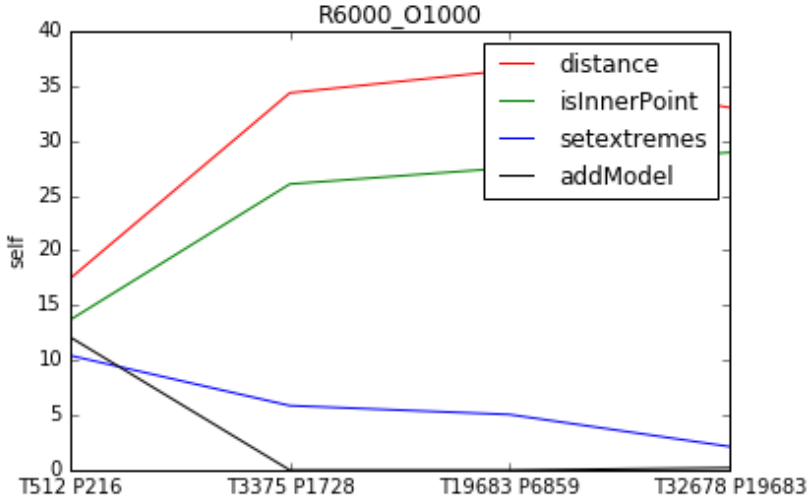

Figura A.67: R6000O1000

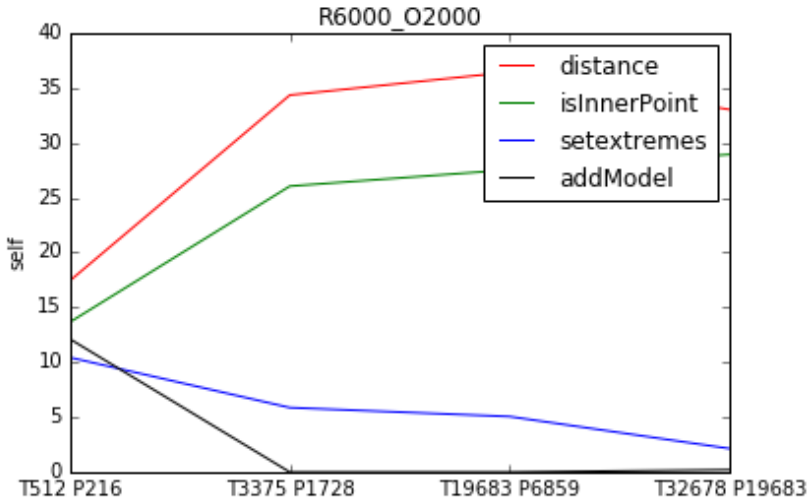

Figura A.68: R6000O2000

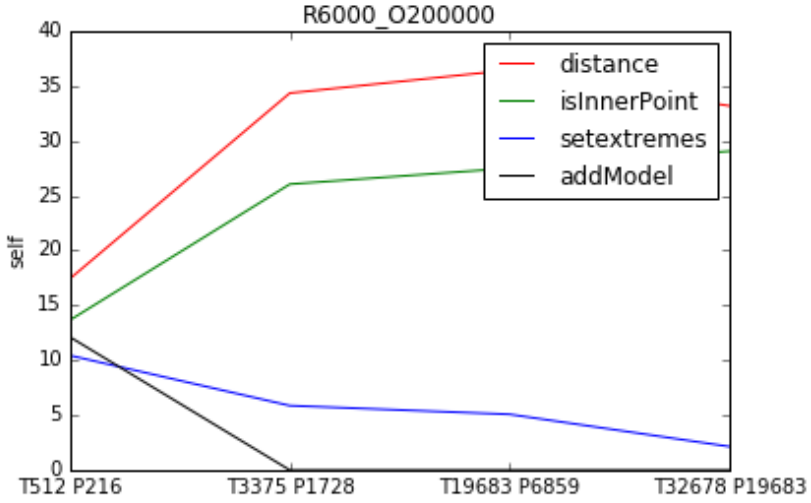

Figura A.69: R6000O200000

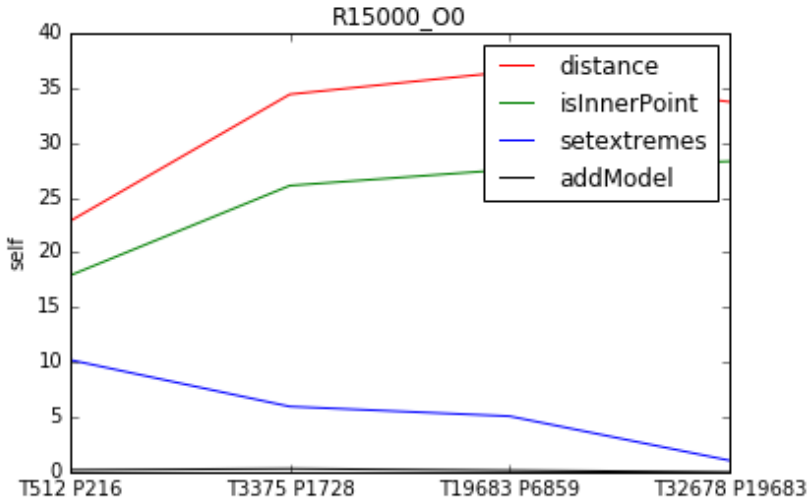

Figura A.70: R15000O0

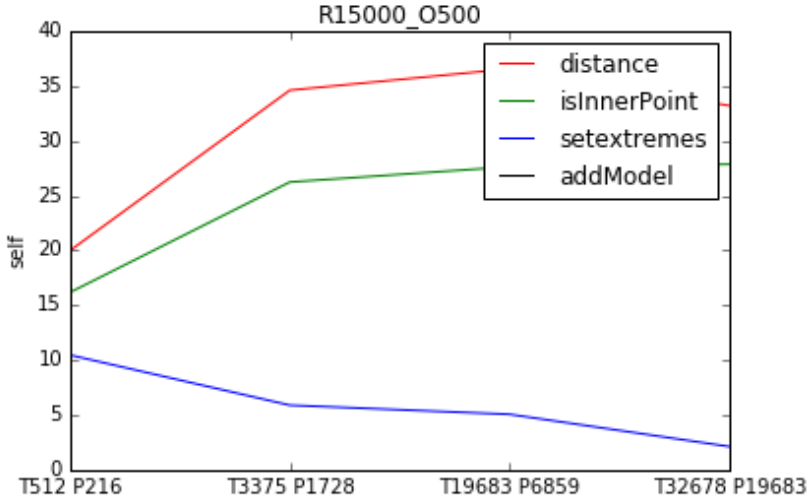

Figura A.71: R15000O500

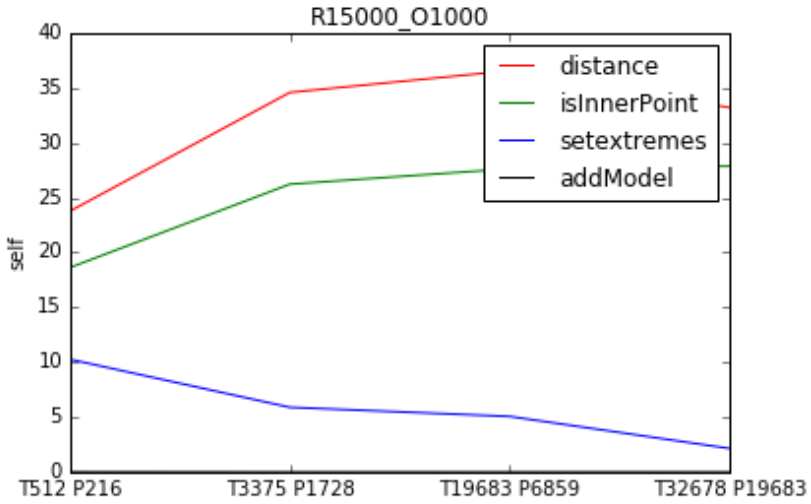

Figura A.72: R15000O1000

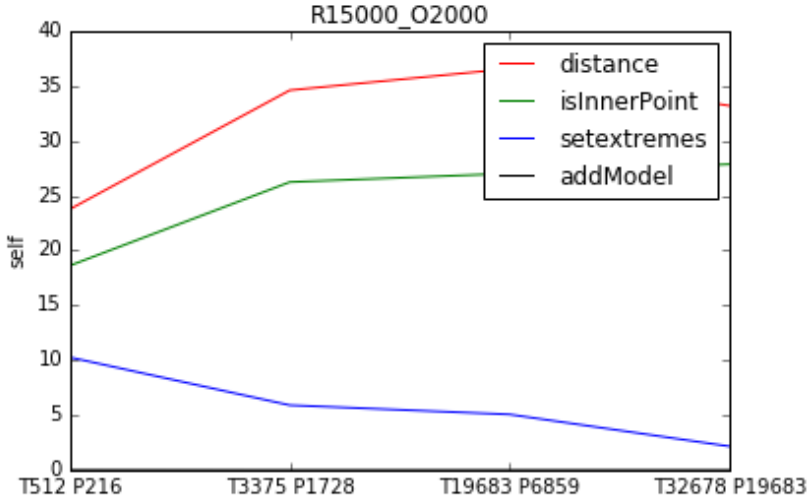

Figura A.73: R15000O2000

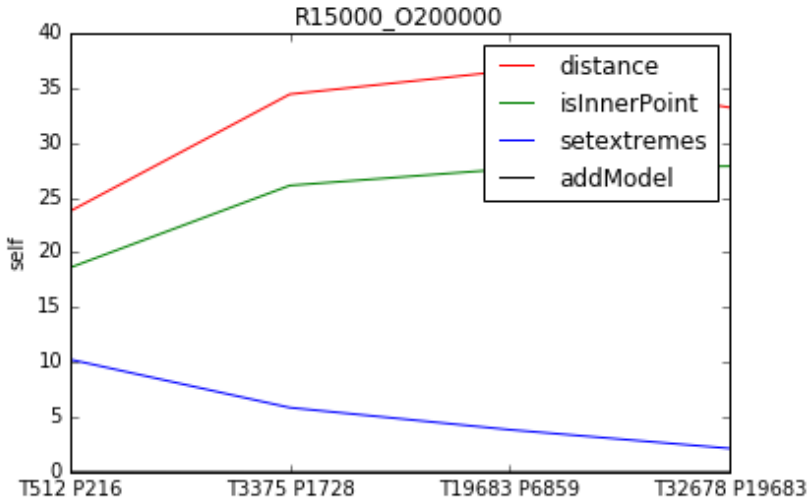

Figura A.74: R15000O200000

Apêndice B

Apêndice 2

Este apêndice inclui um guião de instruções para executar os *scripts* auxiliares criados para facilitar a atividade de analise de desempenho ao BiGGER via *profiling*. O objetivo deste apêndice é garantir a usabilidade dos *scripts* para quem pretender efetuar melhorias de desempenho ao BiGGER. Primeiro constam as instruções para o *script* (DummyJobCreator) que permite a geração dos ficheiros xml fictícios que o BiGGER usa para executar os cenários de *docking* pretendidos. Por fim o apêndice contém uma breve descrição do *script* (ScriptLinearGraphs) que gera gráficos de linhas para os resultados de execução dos testes de *profiling* assim como o que deve ser feito para o utilizar.

B.1 Sobre os *scripts* auxiliares ao *profiling*

B.1.1 DummyJobCreator

O *DummyJobCreator* foi desenvolvido em python e automatiza os dois primeiros passos do procedimento de *profiling* indicado em 4.2. O programa recebe como input um conjunto de argumentos que descrevem o cenário pretendido, gerando como output um ficheiro xml representativo de um *job* com o mesmo formato dos criados pelo *dockprep* para a execução do BiGGER completar. Após a invocação do *script* os seguintes argumentos deverão ser indicados pela ordem referida:

1. probe;

2. sólido geométrico do ligando (cubo/esfera);

3. número de átomos no ligando no caso deste ser uma esfera ou o tamanho do lado se este for cubo;

4. raio de cada átomo presente no ligando (recomendado ser 1.0 para o sólido ser uniforme);
5. target;
6. sólido geométrico do recetor (cubo/esfera);
7. número de átomos no recetor no caso deste ser uma esfera ou o tamanho do lado se for cubo;
8. raio de cada átomo presente no recetor (recomendado ser 1.0 para o sólido ser uniforme);
9. número de rotações;
10. valor da sobreposição mínima.

O programa termina gerando o ficheiro xml na diretoria corrente que poderá ser utilizado pelo BiGGER para executar o cenário de *docking* pretendido.

B.1.2 ScriptLinearGraphs

Este *script* permite a criação dos gráficos de linhas como forma de resultados de *profiling* para todos os cenários a considerar. Apenas é necessária a atualização da matriz de valores de *self cost* no código-fonte. Existe ainda a possibilidade de efetuar o mesmo procedimento para obter gráficos de linhas para o *speedup*, alterando a respetiva matriz. No final da execução deste *script* é gerada uma diretoria de ficheiros onde se podem verificar os gráficos agrupados em três pastas.